Issue #17
November 15, 1998

.....

"The words of the wise are as goads, and as nails fastened by the masters of assemblies." Ecclesiastes 12

"Before criticizing a man I try to walk a mile in his shoes. That way, if he gets mad he's a mile away and barefoot." John Ianetta

.....

BSOUT

For me, fall is a time for reflection. The trees descend into their golden time and then seem to die. And yet, under the surface they are quite alive, and teeming with activity at a smaller, less-visible scale, waiting to burst forwards again in full bloom. I think there's a great metaphor for C= in this. But I have no idea what it is.

In fact things are totally hectic around here, and I haven't given more than a few moments thought towards the 64, so this will be a mighty short editorial. Between a PhD thesis and begging for jobs there hasn't been much 64-time, but with a little here and a little there this issue is finally squeaking out. Everybody worked hard over the summer, and my goal was to get it out in September. Well, you know, these days, if something in the 64 world is only two months late it's doing pretty good, so no big whoop. C=Hacking ought to appear reguarly after December, though.

On the down side, some things, such as the challenge problem from last time, will have to wait until the next issue. I also stayed pretty low-key while putting this issue together, but future issues will be more public in soliciting articles (e.g. on comp.sys.cbm).

In other news, a C64 C-compiler has finally appeared! This outstanding effort is the work of Ullrich von Bassewitz, uz@musoftware.de, the force behind Elite128 among other things. The cc65 webpage is at

http://www.von-bassewitz.de/uz/cc65/

so have a look, and tell Ullrich what a stellar guy he is :). Also, as most of you know, the Chicago Expo took place on October 24, and was a real hoot! Check out

http://driven.c64.org/

for some nice pictures from the Expo, taken by Mark Seelye.

Meanwhile, sit back, relax, and enjoy these latest musings from these, our masters of assembly.

.....
....
..
.

C=H #17

::::::::::::::::::::::::::::::::: Contents :::::::::::::::::::::::::::::::::

BSOUT
o Voluminous ruminations from your unfettered editor.

Jiffies
o Is it a bug or a feature?

The C=Hallenge
o To be continued...

Side Hacking

- "SuperCPU Software Repair", by S. Judd <sjudd@nwu.edu>.
  An amateur's excursion into correcting errant wares.

Main Articles

- "An Optimizing Hybrid LZ77 RLE Data Compression Program, aka Improving Compression Ratio for Low-Resource Decompression", by Pasi 'Albert' Ojala <albert@cs.tut.fi>

  Part two of a two-part article on data compression, giving a detailed description of the compression algorithms used in pucrunch, not to mention the decompression code.

- "VIC-20 Kernel ROM Disassembly Project", by Richard Cini <rcini@email.msn.com>

  This is the first in a series of articles which aims to present a complete, commented disassembly of the VIC-20 ROMs.

- Masters Class: "NTSC/PAL fixing, part I", by Russell Reed <rreed@egypt.org>, Robin Harbron <macbeth@tbaytel.net>, and S. Judd.

  Sit up straight and pay attention. In the Masters Class, a Commodore luminary attempts to instruct a couple of ignorant plebians in his art. In this case, Robin and I set out to learn NTSC/PAL fixing from one of the greats, Decomp/Style. Our first fix, a demo from the obscure Finnish group Pu-239, is included, along with detailed descriptions of our experiences.

- "The Herd Mentality", by Bil Herd <bherd@zeus.jersey.net>

  This is a collection of entertaining musings on Commodore and the development of the C128, as provided by Bil Herd (and that's no bull). If you don't know who Bil Herd _is_, why not type SYS 32800,123,45,6 on a 128 sometime...

..... Credits .....

Editor, The Big Kahuna, The Car'a'carn..... Stephen L. Judd
C=Hacking logo by..... Mark Lawrence

Special thanks to Marko Makela, Olaf Seibert, and the rest of the cbm-hackers for their many otherwise unacknowledged contributions.

Legal disclaimer:
1) If you screw it up it's your own damn fault!
2) If you use someone's stuff without permission you're a dork!

For information on the mailing list, ftp and web sites, send some email to chacking-info@jbrain.com.

About the authors:

Pasi 'Albert' Ojala is a 29 year old software engineer, currently working at a VLSI design company on a RISC DSP core C compiler. Around 1984 a friend introduced him to the VIC-20, and a couple of years later he bought a 64+1541 to replace a broken Spectrum48K. He began writing his own BBS, using ML routines for speed, and later wrote a series of demos under the Pu-239 label. In addition to pucrunch and his many C=Hacking articles, Pasi was written an Amiga 1581 filesystem, a graphics conversion package, a C64 burstloader, and a number of demos. He is currently uses his 64 for hobbyist pursuits, and is contemplating multipart demos for the 64 and the VIC-20, in addition to future C=Hacking articles. Pasi is also a huge Babylon-5 fan, and has a B5 quote page at http://www.cs.tut.fi/~albert/Quotes/B5-quotes.html

Richard Cini is a 31 year old senior loan underwriter who first became involved with Commodore 8-bits in 1981, when his parents bought him a VIC-20 as a birthday present. Mostly he used it for general BASIC programming, with some ML later on, for projects such as controlling the lawn sprinkler system, and for a text-to-speech synthesyzer. All his CBM stuff is packed up right now, along with his other "classic" computers, including a PDP11/34 and a KIM-1. In addition to collecting old computers Richard enjoys gardening, golf, and recently has gotten interested in robotics. As to the C= community, he feels that it

is unique in being fiercely loyal without being evangelical, unlike some other communities, while being extremely creative in making the best use out of the 64.

Robin Harbron is a 26 year old internet tech support at a local independent phone company. He first got involved with C= 8-bits in 1980, playing with school PETs, and in 1983 his parents convinced him to spend the extra money on a C64 instead of getting a VIC-20. Like most of us he played a lot of games, typed in games out of magazines, and tried to write his own games. Now he writes demos, dabbles with Internet stuff, writes C= magazine articles, and, yes, plays games. He is currently working on a few demos and a few games, as well as the "in-progress-but-sometimes-stalled-for-a-real-long-time-until-inspiration-hits-again Internet stuff". He is also working on raising a family, and enjoys music (particularly playing bass and guitar), church, ice hockey and cricket, and classic video games.

..... Jiffies .....

0 REM SHIFT-L

by the cbm.hackers

Everybody knows the old REM shift-L trick in BASIC 2.0, which generates a syntax error upon listing. But why does it work? The answer turns out to be quite interesting.

Normally, when the BASIC interpreter tokenizes a line it strips out characters with the high bit set. One exception is characters within quotes; the other is characters after REM. In those cases, characters are embedded literally into the program line.

Now, BASIC tokens all have the high bit set. When LIST encounters a character with the high bit set, it checks whether it is in quote mode. If it is, the character is outputted as normal. If not, the character is treated as a token, which is expanded by QPLOP (located at $A717). The part of QPLOP which prints keywords looks like this:

```
:LOOP1   DEX              ;Traverse the keyword table
         BEQ :PLOOP
:LOOP2   INY              ;read a keyword
         LDA RESLST,Y
         BPL :LOOP2
         BMI :LOOP1
:PLOOP   INY              ;Print out the keyword
         LDA RESLST,Y
         BMI LISTENT1     ;Exit if on last char
         JSR $AB47        ;Print char, AND #$FF
         BNE :PLOOP
```

The keyword strings in RESLST all stored dextral character inverted (the last char has the high bit set), and the above code just moves forward through the table until it has counted out .X keywords. At that point, :PLOOP prints out the next keyword to the screen, and hops back into LIST.

Shift-L is character code 204, or $CC. When LIST sees this inside of a REM, it sees the high bit set and de-tokenizes it. As it turns out, though, the last token is $CB, which is keyword GO (so that GO TO works). It also turns out that RESLST, the list of BASIC keywords, uses 255 characters. The 256th character is zero (value zero, not character zero).

So, the above code goes through the list, and then prints out token $CC, the first character of which is a null. This zero is sent to $AB47. $AB47 sends it to JSR $FFD2 (which does nothing with the character), performs an AND #$FF, and exits. But that makes the BNE :PLOOP branch _not_ get taken, and the code erroneously moves forwards into... the code which executes the FOR statement!

And the first thing FOR does is look for a valid variable. When you LIST a program, the character immediately following the LIST is a statement terminator (a colon : or else an end of line null). LET (which is called by FOR) reads this character, decides it's an invalid variable name, and generates a ?SYNTAX ERROR.

Because QPLOP uses absolute addressing (LDA RESLST,Y), .Y can wrap around through 255 to traverse the list again. This is why shift-M shift-N etc. all generate valid keywords. Only shift-L will force an error, and it is all due to the zero in the keyword table.

Similar things can happen in other BASICs. In BASIC 4.0, token $DB does the trick. In BASIC 1.0, $CB ought to do it. The problem seems to have been fixed in BASIC 7.0; at least the

trick doesn't seem to work on a 128.

Finally, like most things on the 64, embedding tokens into REM statements can naturally be put to some creative use. For example, RUN once ran a contest where readers submitted stories and riddles and such, which were read by LISTing the program. They were very clever and entertaining, and I close this summary with the one I've remembered since high school:

```
10 REM WHAT'S AN APPLECOSTA?
20 REM {C=-V}T A {INST CTRL-0}E
```

..... The C=Hallenge .....

Wait until next time!

..... Side Hacking .....

SuperCPU software repair
----------> by S. L. Judd

One of the feature articles in this issue deals with NTSC/PAL fixing. But have you ever thought about SCPU fixing? You know how it goes: you have that program that could really benefit from the speed boost, but doesn't work, and usually because of some silly little thing.

Well, it really bugs me to have programs not like my nice hardware for dumb reasons, so I decided I would try my hand at fixing up some programs. The one that really did it for me was the game "Stunt Car Racer" -- I had never played it before, but after getting ahold of it it was clear that here was a game that would be just great with a SuperCPU. I had never done something like this before, but it seemed a doable problem and so I jumped in head first, and this article sums up my inexpert experience to date.

By the way, stuntcar-scpu is totally cool :).

To date I have fixed up just three games: Stunt Car Racer, Rescue on Fractalus, and Stellar 7. My goal was really to "CMD-fix" these programs, to make them run off of my FD-2000 as well as my SCPU. Although these are all games, the techniques should apply equally well to application programs with a bad attitude. Before discussing the fixes, it is probably worthwhile to discuss a few generalities.

I also note that programmers who don't have a SuperCPU might find some of this information helpful in designing their programs to work with SCPUs.

Finally, my fixes are available in the Fridge.

Tools and Process

The tools I used were:

- Action Replay
- Wits
- Paper for taking notes (backs of receipts/envelopes work)

I think this is all that is necessary, although a good sector editor can come in handy for certain things.

After trying a number of different approaches to the problem, the process I've settled on goes roughly like the following:

- Have an idea of what will need fixing
- Familiarize yourself with the program
- Track down the things that need fixing
- Figure out free areas of memory
- Apply patches, and test

Most programs work in more or less the same way: there are some initialization routines, there's a main loop, and there's an interrupt routine or series of routines. The interrupts are easy to find, via the vectors at either $FFFA or at $0314 and friends. The initialization routine can be tougher, but can be deduced from the loader or decompressor; also, some programs point the NMI vector to the reset code, so that RESTORE restarts the program. Finding the

things that need fixing usually involves freezing the program at the appropriate time, and doing a little disassembly. Sometimes a hunt for things like LDA $DC01 is helpful, too. Figuring out free areas of memory is easy, by either getting a good feel for the program, or filling some target memory with a fill byte and then checking it later, to see if it was overwritten. Once the patch works on the 64, all that remains is to test it on the SCPU, and it's all done!

Diagnosis

It seems to me that, at the fundamental level, the SCPU is different from a stock machine in three basic ways: it is a 65816, it runs at 20MHz, and it has hardware registers/different configurations. There are also some strange and mysterious problems that can arise.

All possible opcodes are defined on the 65816, which means that "illegal" or quasi-opcodes will not work correctly. On the 65xx chips, the quasi-opcodes aren't real opcodes -- they are like little holes in the cpu, and things going through those holes fall through different parts of the normal opcode circuitry. Although used by very few programs, a number of copy protection schemes make use of them, so sometimes the program works fine with a SCPU but the copy protection makes it choke -- how very annoying (example: EA's Lords of Conquest). Naturally, disk-based protection methods mean it won't work on an FD-2000, either.

Running at 20Mhz makes all sorts of problems. Any kind of software loop will run too fast -- delay loops, countdown loops, input busy-loops, etc. Also main program loops, so that the game runs unplayably fast (most 3D games never had to worry about being too fast). It can also lead to flickering screens, as we shall see later, and the "play" of some games is designed with 1Mhz in mind -- velocities, accelerations, etc. What looks smooth at the low frame rate might look poor at the high, as we shall also see later. Finally, fastloaders invariably fail at 20Mhz, like any other code using software-based timing.

The SuperCPU also has a series of configuration registers located at $D07x and $D0Bx, which determine things like software speed and VIC optimization modes (which areas of memory are mirrored/copied to the C64's RAM). Note also that enabling hardware registers rearranges $E000 ROM routines. Although it is possible for programs to accidentally reconfigure the SCPU, it is awfully unlikely, since the enable register, which switches the hardware registers in, is sandwiched between disable registers:

$D07D	Hardware register disable
$D07E	Enable hardware registers
$D07F	Hardware register disable

Strangely enough, though, different hardware configurations can sometimes cause problems. For example, newer (v2) SCPUs allow selective mirroring of the stack and zero page, and by default have that mirroring turned OFF. For some totally unknown reason, this caused major problems with an early attempt of mine to fix Stunt Car Racer -- I am told that the old version would slow down to just double-speed, flicker terribly, and more. Turning mirroring back on apparently fixes up the problem. (I have an older SCPU, and hence did not have this problem). So before going after a big fix, it is worthwhile to invest a few minutes in trying different configurations.

Finally, there are other strange problems that can arise. For example, I have two 128s: one is a flat 128, one a 128D. With my 128D, if $D030 is set then the SCPU sometimes -- but not always -- freaks out and locks up. The flat 128 does not have this problem. One reason this is important is that many decompressors INC $D030 to enable 2MHz mode. A simple BIT ($2C) fixes this problem up, but the point is that the SCPU has to interact with the computer, so perhaps that interaction can lead to problems in obscure cases.

Now, if the goal is to CMD-fix the program, there may be a few disk-related things that may need fixing. In addition to stripping out (or possibly fixing up) any fastloaders, most programs annoyingly assume drive #8 is the only drive in town. Also, if the program uses a track-based loader (instead of a file-based loader), then that will need to fixed up as well, and any disk-based copy protection will have to be removed.

There's one other thing to consider, before you fix: is the program really busted? For example, if you've tried a chess program with the SCPU, chances are that you saw no speed improvement. Why not? It turns out that most chess programs use a timer-based search algorithm -- changing the playing strength changes the amount of time the program spends searching, and not the depth of the search. (The reason is to make the gameplay flow a little better -- otherwise

you have very slow play at the beginning, when there are many more moves to consider). So although it might look like it isn't working right with the SCPU, it is actually working quite well.

And that pretty much covers the basic ideas. The first program I fixed up was Stunt Car Racer.

Stunt Car Racer

Stunt Car Racer, in case you don't know, is a 3D driving game, and quite fun. It is also too fast, unplayably fast, at 20MHz. Therefore, it needs to be slowed down!

My first plan... well, suffice to say that most of my original plans were doomed to failure, either from being a bad idea, or from poor implementation. It is clear enough that some sort of delay is needed, though, in the main loop, or perhaps by intercepting the joystick reading routine.

The program has a main loop and an interrupt loop as well. The interrupt handles the display and other things, and all of the game calculations are done in the main loop, which flows like

```
Do some calculations
Draw buffer 1
Swap buffers
Do some calculations
Draw buffer 2
Swap buffers
JMP loop
```

One of my first thoughts was to intercept the joystick I/O, which is easy to find by hunting for LDA $DC01 (or DC00, whichever joystick is used). The patch failed, and possibly because I didn't check that the memory was safe, and possibly because it was in the interrupt routine (I simply don't remember).

Before patching, it is very important to make sure that the patch will survive, and not interfere with the program, so it is very important to find an area of memory that is not used by the program. It took me a little while to figure this out! Finding unused memory was pretty easy -- I just filled the suspect areas with a fill byte, ran the program, and checked that memory. Mapping out the memory areas also aids in saving the file, as un-needed areas don't need to be saved, or can be cleared out to aid compression.

The first free area of memory I found was at $C000. It turns out that this is a sprite, though, and so put some garbage on the screen. The second I tried was $8000, which worked great in practice mode but got overwritten in competition mode -- always test your patches thoroughly! (I had only tested in practice mode). Finally, I found a few little spots in low memory that survived, and placed the patch there. The program does a whole lot of memory moving, and uses nearly all memory. I also left some initialization code at $8000, since it only needed to be run once, at the beginning (to turn on mirroring in v2 SCPUs).

Recall that the main loop has two parts -- one for buffer 1, and one for buffer 2. The trick is to find some code that is common to both sections, like a subroutine call:

```
JSR SomeRoutine
Draw buffer 1
JSR SomeRoutine
Draw buffer 2
```

The patch routine I used was a simple delay loop, redirected from those two JSRs:

```
LDX #$FF
CPX $D012
BNE *-5
DEX
CPX #$FC
BNE *-10
JMP SomeRoutine
```

Of course, this will also slow the program down at 1Mhz; later on I became smarter about my patches, but this one works pretty well.

To save the game and patches, I simply froze it from AR. Just saving from the monitor generally failed; the initialization routine doesn't initialize all I/O settings. Part of the freezing process involves RLE compression, so if you freeze it is a good idea to fill all unused portions of memory -- temporary areas, bitmaps, etc. Another thing to do is to set a freeze point at the init routine,

and then JMP there from the monitor. By clearing the screen, you won't have to look at all the usual freezer ugliness, and at this point freezing isn't any different than saving from the ML monitor and RLE-packing the file. Once saved, I tested a few times from the 64 side, to make sure things worked right.

Whether freezing or saving from the monitor, if the file size is larger than 202 blocks or so, it can't be loaded on the SCPU without a special loader -- unless you compress it first. I naturally recommend using pu-crunch for that purpose, but if you want to do it on the 64 then I recommend using ABCrunch, which works well with the SCPU and gives about as good compression as you can get without an REU.

The result was stuntcar-scpu, which is *awfully* fun when fixed.

Rescue on Fractalus

Next on my list was Rescue on Fractalus, an older (and quite cool) Lucasfilm game that just didn't cut it in the 64 conversion, for a number of reasons (that perhaps could have been avoided). There are at least two versions of the game, one of which doesn't even work on a 128 (good 'ol $D030), but I have the older version, which does work.

With a SuperCPU, though, there are a number of problems. The display flickers terribly. The gameplay is smooth and not at all too fast -- in fact, it is too slow. Specifically, the velocities and turning rates and such do not give a convincing illusion of speed or excitement. The game is copy-protected and uses a track-based fastloader, loaded from disk via B-E, which also saves the high scores to disk. Clearly, this one is a bigger job: the display is too fast, the game constants need adjusting, and the highscore code needs to be replaced by some kernal calls.

The structure of this code is a little different. The main loop handles the (double-buffered) display -- it does all the calculations and draws to the two buffers. The multi-part interrupt loop does the rest -- it swaps buffers, changes the display in different parts of the screen, reads the joystick, and performs the game updates which change your position and direction. It also handles enemies such as saucers, but doesn't handle the bunkers which fire at you from the mountains (the main loop takes care of those).

What does all this mean? First, that the game can be a good ten steps ahead of the screen, which makes things like targeting very difficult. Second, that the bunkers almost never fire at you at 1MHz (they go crazy at 20). Third, that things like velocity and turning rate are rather low, because advancing or turning too quickly would get the game way out of sync (unfortunately, they are still too fast for 1MHz, making targeting difficult and movement clunky). On the other hand, having the movement in the interrupt is the reason that the game does not become unplayably fast at 20MHz, and means that something besides a delay loop is needed.

The interrupt swaps buffers, but the main loop draws them, and because it draws so quickly it can start clearing and drawing to the visible buffer. To make sure this was what I was seeing, I reversed the buffer swap code in the interrupt, so that the drawing buffer was always on-screen. Sure enough, that's what the 20Mhz version looked like.

It turned out to be pretty easy to force the main loop to wait on the interrupt. Although I messed around (unsuccessfully) with intercepting the interrupt loop, the buffer swap code actually modifies a zero-page variable upon swapping. So all the main loop has to do is wait on that variable before charging ahead. I may have made it wait for two frames, because it made the game play a little better.

Now, how to find the velocity and turn code? Well it takes a keypress to change the velocity, so by hunting for LDA $DC01, and tracing back, the routine can be found; at the very least the affected variables may be found, and hunted for. For example, if the result is stored in $D0, then you can search for LDA $D0. The point is to locate the keypress processing code. From there, a little trial and error (setting freeze points and pressing the velocity key) locates the piece of code which deals with changing the velocity, and in particular which variable corresponds to velocity. Finally, from there it just takes another hunt for LDA velocity, ADC velocity, etc. to figure out where the code for updating position and direction is.

In this case, I was pretty sure I had found it, as it went something like

```
LDA velocity
```

```
        LSR
        LSR
        ADC #$20
```

and this was added to the position. To check that this was the code, I just changed the ADC, or removed an LSR, to see that the speed changed. The code for turning left and right and moving up and down was similar, and again after a little trial and error it was clear what code did what. Again, it wasn't necessary to totally understand how these routines worked exactly -- just the general idea of them, in this case to see that a multiple of the velocity was used to change the position and orientation of the player.

So, to fix it up, I just changed that multiple -- probably I NOPed out an LSR above, to basically double the speed, and changed the turning rates similarly. This took a little experimentation, as it not only needed to be playably fast, but also couldn't overflow at high speeds, etc.

But once that was working, all that remained was the highscore table. Finding the table location was pretty easy -- I just got a high score, and while entering my name froze the program, and figured out what got stored where. From there it was pretty easy to figure out what was saved to disk. From the original loader, I also knew where the highscores needed to be loaded to initially (the highscore table gets copied around a lot -- it doesn't just stay at that one location). Figuring out the exact number of bytes to save took a little bit of effort (saving either too many or too few bytes screws it up), but from there it was clear what memory needed to be saved.

So all that remained was to add the usual SETLFS etc. kernal calls, right? Wrong. The program uses all the usual kernal variables (from $90-$C0) for its own purposes. Also recall that I wanted the program to work with device 9, etc. To get around this, I did two things. First, when the program first starts, I save some of the relevant variables to an unused part of memory -- in particular, I save the current drive number. Second, before saving the highscore file, I actually copy all zero page variables from $90-$C2 or so to a temporary location, and then copy them back after saving. That way there are no worries about altering important locations.

Finding memory for the load/save patch was easy -- I just used the area which was previously used for the fastload load/save code. There was enough for the necessary code as well as temporary space for saving the zero page variables.

Finally, I changed some text from Rescue on Fractalus to Behind Jaggi Lines, to distinguish it from the original, and that was that. Works great! And is now more playable and challenging; in short, more the game it always should have been.

Stellar 7

Finally, I tried my hand at Stellar 7. Stellar 7 had several problems. At the main screen, a delay loop tosses you to the mission screen after a while, if no keys are pressed. This is a software loop, and so passes very quickly. The game itself is too fast, so some sort of delay is needed. The mission display is also too fast, and has software delay loops, so that needs fixing. Finally, the game uses kernal calls for loading and saving, but is attached to drive #8; also, my version was split into a bunch of files, and I wanted to cut the number of files down.

Well, by this time it was all pretty straightforward. From the loader, it was easy to figure out which files went where. The mission and main displays were loaded in when needed, and swapped into unused parts of memory when not, so I loaded them in and adjusted the swap variable accordingly -- this left just the highscore and seven level files.

Finding the delay loops was easy -- I just went to the relevant sections of code, froze, and took a look at the loops. There were your basic

```
:LOOP
        LDA $D4         ;Check for keypress
        BMI :key
        DEX
        BNE :LOOP
        DEY
        BNE :LOOP
        DEC counter
        BNE :LOOP
```

```
:key    LDX #$00
        ...
```

Luckily, all routines were pretty much the same as the above. The interrupt routine is in the $0314 vector, and the same routine is used during gameplay.

So the patch is very easy at this point. First, change the IRQ code which does a JMP $EA7B to JMP $CE00

```
. CE00  $EE INC $CFFF
. CE03  $4C JMP $EA7B
```

To fix up the keypress routines, the idea is to change the LDA $D0 into a JSR patch. How to substitute 3 bytes for 2 bytes? The trick is to place the LDX #$00 into the patch routine:

```
. CE06  $20 JSR $CE15           ;Wait for $CFFF
. CE09  $A5 LDA $D4
. CE0B  $10 BPL $CE11
. CE0D  $A2 LDX #$00            ;If key pressed, then LDX #$00
. CE0F  $29 AND #$FF
. CE11  $60 RTS
```

The actual delay is accomplished by waiting on $CFFF:

```
. CE15  $AD LDA $CFFF
. CE18  $C9 CMP #$04
. CE1A  $90 BCC $CE15
. CE1C  $A9 LDA #$00
. CE1E  $8D STA $CFFF
. CE21  $60 RTS
```

As you can see, I waited a (default) of 4 frames. The patch in the game/mission rendering routine works similarly -- I just patched the rendering code to basically JSR $CE15. I also decided to try something new: let the user be able to change that CMP #$04 to make things faster or slower, to suit their tastes. The keyscan values were pretty easy to figure out, so this just required a little patch to check for the "+" and "-" keys, and change location $CE19 accordingly.

Well, that about sums it up. Perhaps if you do some fixing, you might send me a little email describing your own experiences?

```
.......
....
..
.                                   C=H #17
```

::

An Optimizing Hybrid LZ77 RLE Data Compression Program, aka Improving Compression Ratio for Low-Resource Decompression

by Pasi Ojala <albert@cs.tut.fi>

Short:

Pucrunch is a Hybrid LZ77 and RLE compressor, uses an Elias Gamma Code for lengths, mixture of Gamma Code and linear for LZ77 offset, and ranked RLE bytes indexed by the same Gamma Code. Uses no extra memory in decompression.

Introduction

Since I started writing demos for the C64 in 1989 I have always wanted to program a compression program. I had a lot of ideas but never had the time, urge or knowledge to create one. In retrospect, most of the ideas I had then were simply bogus ("magic function theory" as Mark Nelson nicely puts it). But years passed, I gathered more knowledge and finally got an irresistible urge to finally realize my dream.

The nice thing about the delay is that I don't need to write the actual compression program to run on a C64 anymore. I can write it in portable ANSI-C code and just program it to create files that would uncompress themselves when run on a C64. Running the compression program outside of the target system provides at least the following advantages.

* I can use portable ANSI-C code. The compression program can be compiled to run on a Unix box, Amiga, PC etc. And I have all the tools to debug the program and gather profiling information to see why it is so slow :-)
* The program runs much faster than on C64. If it is still slow, there is always multitasking to allow me to do something else while I'm compressing a file.
* There is 'enough' memory available. I can use all the memory I possibly need and use every trick possible to increase the compression ratio as long as the decompression remains possible on a C64.
* Large files can be compressed as easily as shorter files. Most C64 compressors can't handle files larger than around 52-54 kilobytes (210-220 disk blocks).
* Cross-development is easier because you don't have to transfer a file into C64 just to compress it.

Memory Refresh and Terms for Compression

Statistical compression
: Uses the uneven probability distribution of the source symbols to shorten the average code length. Huffman code and arithmetic code belong to this group. By giving a short code to symbols occurring most often, the number of bits needed to represent the symbols decreases. Think of the Morse code for example: the characters you need more often have shorter codes and it takes less time to send the message.

Dictionary compression
: Replaces repeating strings in the source with shorter representations. These may be indices to an actual dictionary (Lempel-Ziv 78) or pointers to previous occurrences (Lempel-Ziv 77). As long as it takes fewer bits to represent the reference than the string itself, we get compression. LZ78 is a lot like the way BASIC substitutes tokens for keywords: one-byte tokens expand to whole words like PRINT#. LZ77 replaces repeated strings with (length,offset) pairs, thus the string VICIICI can be encoded as VICI(3,3) -- the repeated occurrence of the string ICI is replaced by a reference.

Run-length encoding
: Replaces repeating symbols with a single occurrence of the symbol and a repeat count. For example assembly compilers have a .zero keyword or equivalent to fill a number of bytes with zero without needing to list them all in the source code.

Variable-length code
: Any code where the length of the code is not explicitly known but changes depending on the bit values. Some kind of end marker or length count must be provided to make a code a prefix code (uniquely decodable). Compare with ASCII (or Latin-1) text, where you know you get the next letter by reading a full byte from the input. A variable-length code requires you to read part of the data to know how many bits to read next.

Universal codes
: Universal codes are used to encode integer numbers without the need to know the maximum value. Smaller integer values usually get shorter codes. Different universal codes are optimal for different distributions of the values. Universal codes include Elias Gamma and Delta codes, Fibonacci code, and Golomb and Rice codes.

Lossless compression
: Lossless compression algorithms are able to exactly reproduce the original contents unlike lossy compression, which omits details that are not important or perceivable by human sensory system. This article only talks about lossless compression.

My goal in the pucrunch project was to create a compression system in which the decompressor would use minimal resources (both memory and processing power) and still have the best possible compression ratio. A nice bonus would be if it outperformed every other compression program available. These understandingly opposite requirements (minimal resources and good compression ratio) rule out most of the state-of-the-art compression algorithms and in effect only leave RLE and LZ77 to be considered. Another goal was to learn something about data compression and that goal at least has been satisfied.

I started by developing a byte-aligned LZ77+RLE compressor/decompressor and

then added a Huffman backend to it. The Huffman tree took 384 bytes and the code that decoded the tree into an internal representation took 100 bytes. I found out that while the Huffman code gained about 8% in my 40-kilobyte test files, the gain was reduced to only about 3% after accounting the extra code and the Huffman tree.

Then I started a more detailed analysis of the LZ77 offset and length values and the RLE values and concluded that I would get better compression by using a variable-length code. I used a simple variable-length code and scratched the Huffman backend code, as it didn't increase the compression ratio anymore. This version became pucrunch.

Pucrunch does not use byte-aligned data, and is a bit slower than the byte-aligned version because of this, but is much faster than the original version with the Huffman backend attached. And pucrunch still does very well compression-wise. In fact, it does very well indeed, beating even LhA, Zip, and GZip in some cases. But let's not get too much ahead of ourselves.

To get an improvement to the compression ratio for LZ77, we have only a few options left. We can improve on the encoding of literal bytes (bytes that are not compressed), we can reduce the number of literal bytes we need to encode, and shorten the encoding of RLE and LZ77. In the algorithm presented here all these improvement areas are addressed both collectively (one change affects more than one area) and one at a time.

1. By using a variable-length code we can gain compression for even 2-byte LZ77 matches, which in turn reduces the number of literal bytes we need to encode. Most LZ77-variants require 3-byte matches to get any compression because they use so many bits to identify the length and offset values, thus making the code longer than the original bytes would've taken.
2. By using a new literal byte tagging system which distinguishes uncompressed and compressed data efficiently we can reduce number of extra bits needed to make this distinction (the encoding overhead for literal bytes). This is especially important for files that do not compress well.
3. By using RLE in addition to LZ77 we can shorten the encoding for long byte run sequences and at the same time set a convenient upper limit to LZ77 match length. The upper limit performs two functions:
   + we only need to encode integers in a specific range
   + we only need to search strings shorter than this limit (if we find a string long enough, we can stop there)

   Short byte runs are compressed either using RLE or LZ77, whichever gets the best results.
4. By doing statistical compression (more frequent symbols get shorter representations) on the RLE bytes (in this case symbol ranking) we can gain compression for even 2-byte run lengths, which in turn reduces the number of literal bytes we need to encode.
5. By carefully selecting which string matches and/or run lengths to use we can take advantage of the variable-length code. It may be advantageous to compress a string as two shorter matches instead of one long match and a bunch of literal bytes, and it can be better to compress a string as a literal byte and a long match instead of two shorter matches.

This document consists of several parts, which are:
* C64 Considerations - Some words about the target system
* Escape codes - A new tagging system for literal bytes
* File format - What are the primaries that are output
* Graph search - How to squeeze every byte out of this method
* String match - An evolution of how to speed up the LZ77 search
* Some results on the target system files
* Results on the Calgary Corpus Test Suite
* The Decompression routine - 6510 code with commentary

Commodore 64 Considerations

Our target environment (Commodore 64) imposes some restrictions which we have to take into consideration when designing the ideal compression system. A system with a 1-MHz 3-register 8-bit processor and 64 kilobytes of memory certainly imposes a great challenge, and thus also a great sense of achievement for good results.

First, we would like it to be able to decompress as big a program as possible. This in turn requires that the decompression code is located in low memory (most programs that we want to compress start at address 2049) and is as short as possible. Also, the decompression code must not use any extra memory or only very small amounts of it. Extra care must be taken to make certain that the compressed data is not overwritten during the decompression before it has been read.

Secondly, my number one personal requirement is that the basic end address must be correctly set by the decompressor so that the program can be optionally saved in uncompressed form after decompression (although the current decompression code requires that you say "clr" before saving). This also requires that the decompression code is system-friendly, i.e. does not change KERNAL or BASIC variables or other structures. Also, the decompressor shouldn't rely on file size or load end address pointers, because these may be corrupted by e.g. X-modem file transfer protocol (padding bytes may be added).

When these requirements are combined, there is not much selection in where in the memory we can put the decompression code. There are some locations among the first 256 addresses (zeropage) that can be used, the (currently) unused part of the processor stack (0x100..0x1ff), the system input buffer (0x200..0x258) and the tape I/O buffer plus some unused bytes (0x334-0x3ff). The screen memory (0x400..0x7ff) can also be used if necessary. If we can do without the screen memory and the tape buffer, we can potentially decompress files that are located from 0x258 to 0xffff.

The third major requirement is that the decompression should be relatively fast. After 10 seconds the user begins to wonder if the program has crashed or if it is doing anything, even if there is some feedback like border color flashing. This means that the arithmetic used should be mostly 8- or 9-bit (instead of full 16 bits) and there should be very little of it per each decompressed byte. Processor- and memory-intensive algorithms like arithmetic coding and prediction by partial matching (PPM) are pretty much out of the question, and that is saying it mildly. LZ77 seems the only practical alternative. Still, run-length encoding handles long byte runs better than LZ77 and can have a bigger length limit. If we can easily incorporate RLE and LZ77 into the same algorithm, we should get the best features from both.

A part of the decompressor efficiency depends on the format of the compressed data. Byte-aligned codes, where everything is aligned into byte boundaries, can be accessed very quickly; non-byte-aligned variable length codes are much slower to handle, but provide better compression. Note that byte-aligned codes can still have other data sizes than 8. For example you can use 4 bits for LZ77 length and 12 bits for LZ77 offset, which preserves the byte alignment.

The New Tagging System

I call the different types of information my compression algorithm outputs primaries. The primaries in this compression algorithm are:

* literal (uncompressed) bytes and escape sequences,
* LZ77 (length,offset)-pairs,
* RLE (length,byte)-pairs, and
* EOF (end of file marker).

Literal bytes are those bytes that cannot be represented by shorter codes, unlike a part of previously seen data (LZ77), or a part of a longer sequence of the same byte (RLE).

Most compression programs handle the selection between compressed data and literal bytes in a straightforward way by using a prefix bit. If the bit is 0, the following data is a literal byte (uncompressed). If the bit is 1, the following data is compressed. However, this presents the problem that non-compressible data will be expanded from the original 8 bits to 9 bits per byte, i.e. by 12.5 %. If the data isn't very compressible, this overhead consumes all the little savings you may have had using LZ77 or RLE.

Some other data compression algorithms use a value (using variable-length code) that indicates the number of literal bytes that follow, but this is really analogous to a prefix bit, because 1-byte uncompressed data is very common for modestly compressible files. So, using a prefix bit may seem like a good idea, but we may be able to do even better. Let's see what we can come up with. My idea was to somehow use the data itself to mark

compressed and uncompressed data and thus not need any prefix bits.

Let's assume that 75% of the symbols generated are literal bytes. In this case it seems viable to allocate shorter codes for literal bytes, because they are more common than compressed data. This distribution (75% are literal bytes) suggests that we should use 2 bits to determine whether the data is compressed or a literal byte. One of the combinations indicates compressed data, and three of the combinations indicate a literal byte. At the same time those three values divide the literal bytes into three distinct groups. But how do we make the connection between which of the three bit patters we have and what are the literal byte values?

The simplest way is to use a direct mapping. We use two bits (let them be the two most-significant bits) _from the literal bytes themselves_ to indicate compressed data. This way no actual prefix bits are needed. We maintain an escape code (which doesn't need to be static), which is compared to the bits, and if they match, compressed data follows. If the bits do not match, the rest of the literal byte follows. In this way the literal bytes do not expand at all if their most significant bits do not match the escape code, and fewer bits are needed to represent the literal bytes.

Whenever those bits in a literal byte would match the escape code, an escape sequence is generated. Otherwise we could not represent those literal bytes which actually start like the escape code (the top bits match). This escape sequence contains the offending data and a new escape code. This escape sequence looks like

```
# of escape bits    (escape code)
3                   (escape mode select)
# of escape bits    (new escape bits)
8-# of escape bits  (rest of the byte)
= 8 + 3 + # of escape bits
= 13 for 2-bit escapes, i.e. expands the literal byte by 5 bits.
```

Read further to see how we can take advantage of the changing escape code.

You may also remember that in the run-length encoding presented in the previous article two successive equal bytes are used to indicate compressed data (escape condition) and all other bytes are literal bytes. A similar technique is used in some C64 packers (RLE) and crunchers (LZ77), the only difference is that the escape condition is indicated by a fixed byte value. My tag system is in fact an extension to this. Instead of a full byte, I use only a few bits.

We assumed an even distribution of the values and two escape bits, so 1/4 of the values have the same two most significant bits as the escape code. I call this probability that a literal byte has to be escaped the "hit rate". Thus, literal bytes expand in average 25% of the time by 5 bits, making the average length 25% * 13 + 75% * 8 = 9.25. Not surprising, this is longer than using one bit to tag the literal bytes. However, there is one thing we haven't considered yet. The escape sequence has the possibility to change the escape code. Using this feature to its optimum (escape optimization), the average 25% hit rate becomes the -maximum- hit rate.

Also, because the distribution of the literal byte values is seldom flat (some values are more common than others) and there is locality (different parts of the file only contain some of the possible values), from which we can also benefit, the actual hit rate is always much smaller than that. Empirical studies on some test files show that for 2-bit escape codes the actual realized hit rate is only 1.8-6.4%, while the theoretical maximum is the already mentioned 25%.

Previously we assumed the distribution of 75% of literal bytes and 25% of compressed data (other primaries). This prompted us to select 2 escape bits. For other distributions (differently compressible files, not necessarily better or worse) some other number of escape bits may be more suitable. The compressor tries different number of escape bits and select the value which gives the best overall results. The following table summarizes the hit rates on the test files for different number of escape bits.

1-bit	2-bit	3-bit	4-bit	File
50.0%	25.0%	12.5%	6.250%	Maximum
25.3%	2.5%	0.3%	0.090%	ivanova.bin
26.5%	2.4%	0.8%	0.063%	sheridan.bin
20.7%	1.8%	0.2%	0.041%	delenn.bin
26.5%	6.4%	2.5%	0.712%	bs.bin
9.06	8.32	8.15	8.050	bits/Byte for bs.bin

As can be seen from the table, the realized hit rates are dramatically

smaller than the theoretical maximum values. A thought might occur that we should always select 4-bit (or longer) escapes, because it reduces the hit rate and presents the minimum overhead for literal bytes. Unfortunately increasing the number of escape bits also increases the code length of the compressed data. So, it is a matter of finding the optimum setting.

If there are very few literal bytes compared to other primaries, 1-bit escape or no escape at all gives very short codes to compressed data, but causes more literal bytes to be escaped, which means 4 bits extra for each escaped byte (with 1-bit escapes). If the majority of primaries are literal bytes, for example a 6-bit escape code causes most of the literal bytes to be output as 8-bit codes (no expansion), but makes the other primaries 6 bits longer. Currently the compressor automatically selects the best number of escape bits, but this can be overridden by the user with the -e option.

The cases in the example with 1-bit escape code validates the original suggestion: use a prefix bit. A simple prefix bit would produce better results on three of the previous test files (although only slightly). For delenn.bin (1 vs 0.828) the escape system works better. On the other hand, 1-bit escape code is not selected for any of the files, because 2-bit escape gives better overall results.

-Note:- for 7-bit ASCII text files, where the top bit is always 0 (like most of the Calgary Corpus files), the hit rate is 0% for even 1-bit escapes. Thus, literal bytes do not expand at all. This is equivalent to using a prefix bit and 7-bit literals, but does not need separate algorithm to detect and handle 7-bit literals.

For Calgary Corpus files the number of tag bits per primary (counting the escape sequences and other overhead) ranges from as low as 0.46 (book1) to 1.07 (geo) and 1.09 (pic). These two files (geo and pic) are the only ones in the suite where a simple prefix bit would be better than the escape system. The average is 0.74 tag bits per primary.

In Canterbury Corpus the tag bits per primary ranges from 0.44 (plrabn12.txt) to 1.09 (ptt5), which is the only one above 0.85 (sum). The average is 0.61 tag bits per primary.

Primaries Used for Compression

The compressor uses the previously described escape-bit system while generating its output. I call the different groups of bits that are generated primaries, whether it is the correct term or not. You are welcome to suggest a better term for them. The primaries in this compression algorithm are: literal byte (and escape sequence), LZ77 (length,offset)-pair, RLE (length, byte)-pair, and EOF (end of file marker).

If the top bits of a literal byte do not match the escape code, the byte is output as-is. If the bits match, an escape sequence is generated, with the new escape code. Other primaries start with the escape code.

The Elias Gamma Code is used extensively. This code consists of two parts: a unary code (a one-bit preceded by zero-bits) and a binary code part. The first part tells the decoder how many bits are used for the binary code part. Being a universal code, it produces shorter codes for small integers and longer codes for larger integers. Because we expect we need to encode a lot of small integers (there are more short string matches and shorter equal byte runs than long ones), this reduces the total number of bits needed. See the previous article for a more in-depth delve into statistical compression and universal codes. To understand this article, you only need to keep in mind that small integer value equals short code. The following discusses the encoding of the primaries.

The most frequent compressed data is LZ77. The length of the match is output in Elias Gamma code, with "0" meaning the length of 2, "100" length of 3, "101" length of 4 and so on. If the length is not 2, a LZ77 offset value follows. This offset takes 9 to 22 bits. If the length is 2, the next bit defines whether this is LZ77 or RLE/Escape. If the bit is 0, an 8-bit LZ77 offset value follows. (Note that this restricts the offset for 2-byte matches to 1..256.) If the bit is 1, the next bit decides between escape (0) and RLE (1).

The code for an escape sequence is thus e..e010n..ne....e, where E is the byte, and N is the new escape code. Example:

```
  * We are using 2-bit escapes
  * The current escape code is "11"
  * We need to encode a byte 0xca == 0b11001010
  * The escape code and the byte high bits match (both are "11")
  * We output the current escape code "11"
  * We output the escaped identification "010"
  * We output the new escape bits, for example "10" (depends on escape
    optimization)
  * We output the rest of the escaped byte "001010"
  * So, we have output the string "1101010001010"
```

When the decompressor receives this string of bits, it finds that the first two bits match with the escape code, it finds the escape identification ("010") and then gets the new escape, the rest of the original byte and combines it with the old escape code to get a whole byte.

The end of file condition is encoded to the LZ77 offset and the RLE is subdivided into long and short versions. Read further, and you get a better idea about why this kind of encoding is selected.

When I studied the distribution of the length values (LZ77 and short RLE lengths), I noticed that the smaller the value, the more occurrences. The following table shows an example of length value distribution.

	LZLEN	S-RLE
2	1975	477
3-4	1480	330
5-8	492	166
9-16	125	57
17-32	31	33
33-64	8	15

The first column gives a range of values. The first entry has a single value (2), the second two values (3 and 4), and so on. The second column shows how many times the different LZ77 match lengths are used, the last column shows how many times short RLE lengths are used. The distribution of the values gives a hint of how to most efficiently encode the values. We can see from the table for example that values 2-4 are used 3455 times, while values 5-64 are used only 656 times. The more common values need to get shorter codes, while the less-used ones can be longer.

Because in each "magnitude" there are approximately half as many values than in the preceding one, it almost immediately occurred to me that the optimal way to encode the length values (decremented by one) is:

Value	Encoding	Range	Gained
0000000	not possible		
0000001	0	1	-6 bits
000001x	10x	2-3	-4 bits
00001xx	110xx	4-7	-2 bits
0001xxx	1110xxx	8-15	+0 bits
001xxxx	11110xxxx	16-31	+2 bits
01xxxxx	111110xxxxx	32-63	+4 bits
1xxxxxx	111111xxxxxx	64-127	+5 bits

The first column gives the binary code of the original value (with x denoting 0 or 1, xx 0..3, xxx 0..7 and so on), the second column gives the encoding of the value(s). The third column lists the original value range in decimal notation.

The last column summarizes the difference between this code and a 7-bit binary code. Using the previous encoding for the length distribution presented reduces the number of bits used compared to a direct binary representation considerably. Later I found out that this encoding in fact is Elias Gamma Code, only the assignment of 0- and 1-bits in the prefix is reversed, and in this version the length is limited. Currently the maximum value is selectable between 64 and 256.

So, to recap, this version of the Gamma code can encode numbers from 1 to 255 (1 to 127 in the example). LZ77 and RLE lengths that are used start from 2, because that is the shortest length that gives us any compression. These length values are first decremented by one, thus length 2 becomes "0", and for example length 64 becomes "11111011111".

The distribution of the LZ77 offset values (pointer to a previous occurrence of a string) is not at all similar to the length distribution. Admittedly, the distribution isn't exactly flat, but it also isn't as radical as the length value distribution either. I decided to encode the lower 8 bits (automatically selected or user-selectable between 8 and 12 bits in the current version) of the offset as-is (i.e. binary code) and the upper part with my version of the Elias Gamma Code. However, 2-byte

matches always have an 8-bit offset value. The reason for this is discussed shortly.

Because the upper part can contain the value 0 (so that we can represent offsets from 0 to 255 with a 8-bit lower part), and the code can't directly represent zero, the upper part of the LZ77 offset is incremented by one before encoding (unlike the length values which are decremented by one). Also, one code is reserved for an end of file (EOF) symbol. This restricts the offset value somewhat, but the loss in compression is negligible.

With the previous encoding 2-byte LZ77 matches would only gain 4 bits (with 2-bit escapes) for each offset from 1 to 256, and 2 bits for each offset from 257 to 768. In the first case 9 bits would be used to represent the offset (one bit for gamma code representing the high part 0, and 8 bits for the low part of the offset), in the latter case 11 bits are used, because each "magnitude" of values in the Gamma code consumes two more bits than the previous one.

The first case (offset 1..256) is much more frequent than the second case, because it saves more bits, and also because the symbol source statistics (whatever they are) guarantee 2-byte matches in recent history (much better chance than for 3-byte matches, for example). If we restrict the offset for a 2-byte LZ77 match to 8 bits (1..256), we don't lose so much compression at all, but instead we could shorten the code by one bit. This one bit comes from the fact that before we had to use one bit to make the selection "8-bit or longer". Because we only have "8-bit" now, we don't need that select bit anymore.

Or, we can use that select bit to a new purpose to select whether this code really is LZ77 or something else. Compared to the older encoding (which I'm not detailing here, for clarity's sake. This is already much too complicated to follow, and only slightly easier to describe) the codes for escape sequence, RLE and End of File are still the same length, but the code for LZ77 has been shortened by one bit. Because LZ77 is the most frequently used primary, this presents a saving that more than compensates for the loss of 2-byte LZ77 matches with offsets 257..768 (which we can no longer represent, because we fixed the offset for 2-byte matches to use exactly 8 bits).

Run length encoding is also a bit revised. I found out that a lot of bits can be gained by using the same length encoding for RLE as for LZ77. On the other hand, we still should be able to represent long repeat counts as that's where RLE is most efficient. I decided to split RLE into two modes:

  * short RLE for short (e.g. 2..128) equal byte strings
  * long RLE for long equal byte strings

The Long RLE selection is encoded into the Short RLE code. Short RLE only uses half of its coding space, i.e. if the maximum value for the gamma code is 127, short RLE uses only values 1..63. Larger values switches the decoder into Long RLE mode and more bits are read to complete the run length value.

For further compression in RLE we rank all the used RLE bytes (the values that are repeated in RLE) in the decreasing probability order. The values are put into a table, and only the table indices are output. The indices are also encoded using a variable length code (the same gamma code, surprise..), which uses less bits for smaller integer values. As there are more RLE's with smaller indices, the average code length decreases. In decompression we simply get the gamma code value and then use the value as an index into the table to get the value to repeat.

Instead of reserving full 256 bytes for the table we only put the top 31 RLE bytes into the table. Normally this is enough. If there happens to be a byte run with a value not in the table we use a similar technique as for the short/long RLE selection. If the table index is larger than 31, it means we don't have the value in the table. We use the values 32..63 to select the 'escaped' mode and simultaneously send the 5 most significant bits of the value (there are 32 distinct values in the range 32..63). The rest 3 bits of the byte are sent separately.

If you are more than confused, forget everything I said in this chapter and look at the decompression pseudo-code later in this article.

Graph Search - Selecting Primaries

In free-parse methods there are several ways to divide the file into parts,

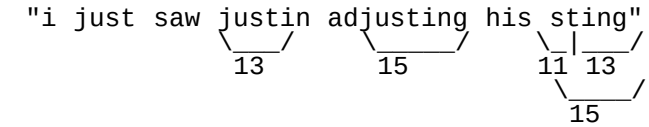
each of which is equally valid but not necessary equally efficient in terms of compression ratio.

```
     "i just saw justin adjusting his sting"

     "i just saw", (-9,4), "in ad", (-9,6), "g his", (-25,2), (-10,4)
     "i just saw", (-9,4), "in ad", (-9,6), "g his ", (-10,5)
```

The latter two lines show how the sentence could be encoded using literal bytes and (offset, length) pairs. As you can see, we have two different encodings for a single string and they are both valid, i.e. they will produce the same string after decompression. This is what free-parse is: there are several possible ways to divide the input into parts. If we are clever, we will of course select the encoding that produces the shortest compressed version. But how do we find this shortest version? How does the data compressor decide which primary to generate in each step?

The most efficient way the file can be divided is determined by a sort of a graph-search algorithm, which finds the shortest possible route from the start of the file to the end of the file. Well, actually the algorithm proceeds from the end of the file to the beginning for efficiency reasons, but the result is the same anyway: the path that minimizes the bits emitted is determined and remembered. If the parameters (number of escape bits or the variable length codes or their parameters) are changed, the graph search must be re-executed.



Think of the string as separate characters. You can jump to the next character by paying 8 bits to do so (not shown in the figure), unless the top bits of the character match with the escape code (in which case you need more bits to send the character "escaped"). If the history buffer contains a string that matches a string starting at the current character you can jump over the string by paying as many bits as representing the LZ77 (offset,length)-pair takes (including escape bits), in this example from 11 to 15 bits. And the same applies if you have RLE starting at the character. Then you just find the least-expensive way to get from the start of the file to the end and you have found the optimal encoding. In this case the last characters " sting" would be optimally encoded with 8(literal " ") + 15("sting") = 23 instead of 11(" s") + 13("ting") = 24 bits.

The algorithm can be written either cleverly or not-so. We can take a real short-cut compared to a full-blown graph search because we can/need to only go forwards in the file: we can simply start from the end! Our accounting information which is updated when we pass each location in the data consists of three values:

1. the minimum bits from this location to the end of file.
2. the mode (literal, LZ77 or RLE) to use to get that minimum
3. the "jump" length for LZ77 and RLE

For each location we try to jump forward (to a location we already processed) one location, LZ77 match length locations (if a match exists), or RLE length locations (if equal bytes follow) and select the shortest route, update the tables accordingly. In addition, if we have a LZ77 or RLE length of for example 18, we also check jumps 17, 16, 15, ... This gives a little extra compression. Because we are doing the "tree traverse" starting from the "leaves", we only need to visit/process each location once. Nothing located after the current location can't change, so there is never any need to update a location.

To be able to find the minimal path, the algorithm needs the length of the RLE (the number of the identical bytes following) and the maximum LZ77 length/offset (an identical string appearing earlier in the file) for each byte/location in the file. This is the most time-consuming -and-memory-consuming part of the compression. I have used several methods to make the search faster. See String Match Speedup later in this article. Fortunately these searches can be done first, and the actual optimization can use the cached values.

Then what is the rationale behind this optimization? It works because you are not forced to take every compression opportunity, but select the best ones. The compression community calls this "lazy coding" or "non-greedy" selection. You may want to emit a literal byte even if there is a 2-byte LZ77 match, because in the next position in the file you may have a longer match. This is actually more complicated than that, but just take my word for it that there is a difference. Not a very big difference, and only

significant for variable-length code, but it is there and I was after every last bit of compression, remember.

Note that the decision-making between primaries is quite simple if a fixed-length code is used. A one-step lookahead is enough to guarantee optimal parsing. If there is a more advantageous match in the next location, we output a literal byte and that longer match instead of the shorter match. I don't have time or space here to go very deeply on that, but the main reason is that in fixed-length code it doesn't matter whether you represent a part of data as two matches of lengths 2 and 8 or as matches of lengths 3 and 7 or as any other possible combination (if matches of those lengths exist). This is not true for a variable-length code and/or a statistical compression backend. Different match lengths and offsets no longer generate equal-length codes.

Note also that most LZ77 compression algorithms need at least 3-byte match to break even, i.e. not expanding the data. This is not surprising when you stop to think about it. To gain something from 2-byte matches you need to encode the LZ77 match into 15 bits. This is very little. A generic LZ77 compressor would use one bit to select between a literal and LZ77, 12 bits for moderate offset, and you have 2 bits left for match length. I imagine the rationale to exclude 2-byte matches also include "the potential savings percentage for 2-byte matches is insignificant". Pucrunch gets around this by using the tag system and Elias Gamma Code, and does indeed gain bits from even 2-byte matches.

After we have decided on what primaries to output, we still have to make sure we get the best results from the literal tag system. Escape optimization handles this. In this stage we know which parts of the data are emitted as literal bytes and we can select the minimal path from the first literal byte to the last in the same way we optimized the primaries. Literal bytes that match the escape code generate an escape sequence, thus using more bits than unescaped literal bytes, and we need to minimize these occurrences.

For each literal byte there is a corresponding new escape code which minimizes the path to the end of the file. If the literal byte's high bits match the current escape code, this new escape code is used next. The escape optimization routine proceeds from the end of the file to the beginning like the graph search, but it proceeds linearly and is thus much faster.

I already noted that the new literal byte tagging system exploits the locality in the literal byte values. If there is no correlation between the bytes, the tagging system does not do well at all. Most of the time, however, the system works very well, performing 50% better than the prefix-bit approach.

The escape optimization routine is currently very fast. A little algorithmic magic removed a lot of code from the original version. A fast escape optimization routine is quite advantageous, because the number of escape bits can now vary from 0 (uncompressed bytes always escaped) to 8 and we need to run the routine again if we change the number of escape bits used to select the optimal escape code changes.

Because escaped literal bytes actually expand the data, we need a safety area, or otherwise the compressed data may get overwritten by the decompressed data before we have used it. Some extra bytes need to be reserved for the end of file marker. The compression routine finds out how many bytes we need for safety buffer by keeping track of the difference between input and output sizes while creating the compressed file.

```
            $1000 .. $2000
            |OOOOOOOO|           O=original file

        $801 ..
        |D|CCCCC|                C=compressed data (D=decompressor)

 $f7..      $1000     $2010
 |D|            |CCCCC|          Before decompression starts
            ^    ^
            W    R               W=write pointer, R=read pointer
```

If the original file is located at $1000-$1fff, and the calculated safety area is 16 bytes, the compressed version will be copied by the decompression routine higher in memory so that the last byte is at $200f. In this way, the minimum amount of other memory is overwritten by the decompression. If the safety are would exceed the top of memory, we need a wrap buffer. This is handled automatically by the compressor. The read pointer wraps from the end of memory to the wrap buffer, allowing the original file to extend up to the end of the memory, all the way to $ffff.

You can get the compression program to tell you which memory areas it uses by specifying the "-s" option. Normally the safety buffer needed is less than a dozen bytes.

To sum things up, Pucrunch operates in several steps:
1. Find RLE and LZ77 data, pre-select RLE byte table
2. Graph search, i.e. which primaries to use
3. Primaries/Literal bytes ratio decides how many escape bits to use
4. Escape optimization, which escape codes to use
5. Update RLE ranks and the RLE byte table
6. Determine the safety area size and output the file.

String Match Speedup

To be able to select the most efficient combination of primaries we of course first need to find out what kind of primaries are available for selection. If the file doesn't have repeated bytes, we can't use RLE. If the file doesn't have repeating byte strings, we can't use LZ77. This string matching is the most time-consuming operation in LZ77 compression simply because of the amount of the comparison operations needed. Any improvement in the match algorithm can decrease the compression time considerably. Pucrunch is a living proof on that.

The RLE search is straightforward and fast: loop from the current position (P) forwards in the file counting each step until a different-valued byte is found or the end of the file is reached. This count can then be used as the RLE byte count (if the graph search decides to use RLE). The code can also be optimized to initialize counts for all locations that belonged to the RLE, because by definition there are only one-valued bytes in each one. Let us mark the current file position by P.

```
unsigned char *a = indata + P, val = *a++;
int top = inlen - P;
int rlelen = 1;

/* Loop for the whole RLE */
while(rlelen<top && *a++ == val)
    rlelen++;

for(i=0;i<rlelen-1;i++)
    rle[P+i] = rlelen-i;
```

With LZ77 we can't use the same technique as for RLE (i.e. using the information about current match to skip subsequent file locations to speed up the search). For LZ77 we need to find the longest possible, and -nearest- possible, string that matches the bytes starting from the current location. The nearer the match, the less bits are needed to represent the offset from the current position.

Naively, we could start comparing the strings starting from P-1 and P, remembering the length of the matching part and then doing the same at P-2 and P, P-3 and P, .. P-j and P (j is the maximum search offset). The longest match and its location (offset from the current position) are then remembered and initialized. If we find a match longer or equal than the maximum length we can actually use, we can stop the search there. (The code used to represent the length values may have an upper limit.)

This may be the first implementation that comes to your (and my) mind, and might not seem so bad at first. In reality, it is a very slow way to do the search: the -Brute Force- method. It could take somewhere about (n^3) byte compares to process a file of the length n (a mathematically inclined person would probably give a better estimate). However, using the already determined RLE value to our advantage permits us to rule out the worst-case projection, which happens when all bytes are the same value. We only search LZ77 matches if the current file position has shorter RLE sequence than the maximum LZ77 copy length.

The first thing I did to improve the speed is to remember the position where each byte has last been seen. A simple 256-entry table handles that. Using this table, the search can directly start from the first potential match, and we don't need to search for it byte-by-byte anymore. The table is continually updated when we move toward to the end of the file.

That didn't give much of an improvement, but then I increased the table to 256*256 entries, making it possible to locate the latest occurrence of any byte -pair- instead. The table indexed with the byte values and the table

contents directly gives the position in file where these two bytes were last seen. Because the shortest possible string that would offer any compression (for my encoding of LZ77) is two bytes long, this byte-pair history is very suitable indeed. Also, the first (shortest possible, i.e. 2-byte) match is found directly from the byte-pair history. This gave a moderate 30% decrease in compression time for one of my test files (from 28 minutes to 17 minutes on a 25 MHz 68030).

The second idea was to quickly discard the strings that had no chance of being longer matches than the one already found. A one-byte hash value (sort of a checksum here, it is never used to index a hash table in this algorithm, but I rather use "hash value" than "checksum") is calculated from each three bytes of data. The values are calculated once and put into a table, so we only need two memory fetches to know if two 3-byte strings are different. If the hash values are different, at least one of the data bytes differ. If the hash values are equal, we have to compare the original bytes. The hash values of the strategic positions of the strings to compare are then .. compared. This strategic position is the location two bytes earlier than the longest match so far. If the hash values differ, there is no chance that the match is longer than the current one. It may be not even be as long, because one of the two earlier bytes may be different. If the hash values are equal, the brute-force byte-by-byte compare has to be done. However, the hash value check already discards a huge number of candidates and more than generously pays back its own memory references. Using the hash values the compression time shortens by 50% (from 17 minutes to 8 minutes).

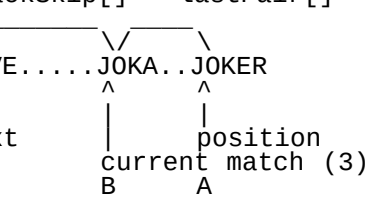
Okay, the byte-pair table tells us where the latest occurrence of any byte pair is located. Still, for the latest occurrence before -that- one we have to do a brute force search. The next improvement was to use the byte-pair table to generate a linked list of the byte pairs with the same value. In fact, this linked list can be trivially represented as a table, using the same indexing as the file positions. To locate the previous occurrence of a 2-byte string starting at location P, look at backSkip[P].

```
 /* Update the two-byte history & backSkip */
 if(P+1<inlen)
 {
     int index = (indata[P]<<8) | indata[P+1];

     backSkip[P] = lastPair[index];
     lastPair[index] = P+1;
 }
```

Actually the values in the table are one bigger than the real table indices. This is because the values are of type unsigned short (can only represent non-negative values), and I wanted zero to mean "not occurred".

This table makes the search of the next (previous) location to consider much faster, because it is a single table reference. The compression time was reduced from 6 minutes to 1 minute 10 seconds. Quite an improvement from the original 28 minutes!

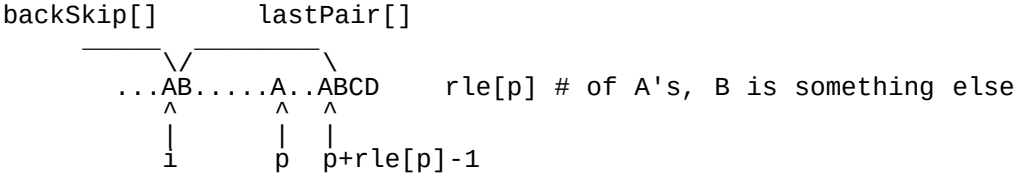


In this example we are looking at the string "JOKER" at location A. Using the lastPair[] table (with the index "JO", the byte values at the current location A) we can jump directly to the latest match at B, which is "JO", 2 bytes long. The hash values for the string at B ("JOK") and at A ("JOK") are compared. Because they are equal, we have a potential longer match (3 bytes), and the strings "JOKE.." and "JOKA.." are compared. A match of length 3 is found (the 4th byte differs). The backSkip[] table with the index B gives the previous location where the 2-byte string "JO" can be found, i.e. C. The hash value for the strategic position of the string in the current position A ("OKE") is then compared to the hash value of the corresponding position in the next potential match starting at C ("OVE"). They don't match, so the string starting at C ("JOVE..") can't include a longer match than the current longest match at B.

There is also another trick that takes advantage of the already determined RLE lengths. If the RLE lengths for the positions to compare don't match, we can directly skip to the next potential match. Note that the RLE bytes (the data bytes) are the same, and need not be compared, because the first

byte (two bytes) are always equal on both positions (our backSkip[] table guarantees that). The RLE length value can also be used to skip the start of the strings when comparing them.

Another improvement to the search code made it dramatically faster than before on highly redundant files (such as pic from the Calgary Corpus Suite, which was the Achilles' heel until then). Basically the new search method just skips over the RLE part (if any) in the search position and then checks if the located position has equal number (and value) of RLE bytes before it.



The algorithm searches for a two-byte string which starts at p + rle[p]-1, i.e. the last rle byte ('A') and the non-matching one ('B'). When it finds such location (simple lastPair[] or backSkip[] lookup), it checks if the rle in the compare position (i-(rle[p]-1)) is long enough (i.e. the same number of A's before the B in both places). If there are, the normal hash value check is performed on the strings and if it succeeds, the brute-force byte-compare is done.

The rationale behind this idea is quite simple. There are dramatically less matches for "AB" than for "AA", so we get a huge speedup with this approach. We are still guaranteed to find the most recent longest match there is.

Note that a compression method similar to RLE can be realized using just LZ77. You just emit the first byte as a literal byte, and output a LZ77 code with offset 1 and the original RLE length minus 1. You can thus consider RLE as a special case, which offers tighter encoding of the necessary information. Also, as my LZ77 limits the copy size to 64/128/256 bytes, a RLE version providing lengths up to 32 kilobytes is a big improvement, even if the code for it is somewhat longer.

The Decompression Routine

Any lossless compression program is totally useless unless there exists a decompression program which takes in the compressed file and -- using only that information -- generates the original file exactly. In this case the decompression program must run on C64's 6510 microprocessor, which had its impact on the algorithm development also. Regardless of the algorithm, there are several requirements that the decompression code must satisfy:

1. Correctness - the decompression must behave accurately to guarantee lossless decompression
2. Memory usage - the less memory is used the better
3. Speed - fast decompression is preferred to slower one

The latter two requirements can be and are complementary. A somewhat faster decompression for the same algorithm is possible if more memory can be used (although in this case the difference is quite small). In any case the correctness of the result is the most important thing.

A short pseudo-code of the decompression algorithm follows before I go to the actual C64 decompression code.

```
copy the decompression code to low memory
copy the compressed data forward in memory so that it isn't
  overwritten before we have read it (setup safety & wrap buffers)
setup source and destination pointers
initialize RLE byte code table, the number of escape bits etc.
set initial escape code
do forever
    get the number of escape bits "bits"
    if "bits" do not match with the escape code
        read more bits to complete a byte and output it
    else
        get Elias Gamma Code "value" and add 1 to it
        if "value" is 2
            get 1 bit
            if bit is 0
```

```
                    it is 2-byte LZ77
                    get 8 bits for "offset"
                    copy 2 bytes from "offset" bytes before current
                        output position into current output position
                else
                    get 1 bit
                    if bit is 0
                        it is an escaped literal byte
                        get new escape code
                        get more bits to complete a byte with the
                            current escape code and output it
                        use the new escape code
                    else
                        it is RLE
                        get Elias Gamma Code "length"
                        if "length" larger or equal than half the maximum
                            it is long RLE
                            get more bits to complete a byte "lo"
                            get Elias Gamma Code "hi", subtract 1
                            combine "lo" and "hi" into "length"
                        endif
                        get Elias Gamma Code "index"
                        if "index" is larger than 31
                            get 3 more bits to complete "byte"
                        else
                            get "byte" from RLE byte code table from
                                index "index"
                        endif
                        copy "byte" to the output "length" times
                    endif
                endif
            else
                it is LZ77
                get Elias Gamma Code "hi" and subtract 1 from it
                if "hi" is the maximum value - 1
                    end decompression and start program
                endif
                get 8..12 bits "lo" (depending on settings)
                combine "hi" and "lo" into "offset"
                copy "value" number of bytes from "offset" bytes before
                    current output position into current output position
            endif
        endif
    end do
```

The following routine is the pucrunch decompression code. The code runs on the C64 or C128's C64-mode and a modified version is used for Vic20 and C16/Plus4. It can be compiled by at least DASM V2.12.04. Note that the compressor automatically attaches this code to the packet and sets the different parameters to match the compressed data. I will insert additional commentary between strategic code points in addition to the comments that are already in the code.

Note that at this point it is only possible to make the decompression code shorter by removing features. At least I think that it is now so. If I'm wrong, feel free to point it out to me. Tim Rogers <timr@eurodltd.co.uk> did manage to snip off 2 bytes, thanks! However, there are some features you may consider unnecessary. The code can be shortened by:

* No basic end address set: 8 bytes
* No 2 MHz mode set/reset: 6 bytes
* No wrap option: 12 bytes

Actually, if the wrap option is not used, the compressor automatically selects the shorter decompression code (only for the C64 version).

```
        processor 6502

BASEND  EQU $2d         ; start of basic variables (updated at EOF)
LZPOS   EQU $2d         ; temporary, BASEND *MUST* *BE* *UPDATED* at EOF

bitstr  EQU $f7         ; Hint the value beforehand
xstore  EQU $c3         ; tape load temp

WRAPBUF EQU $004b       ; 'wrap' buffer, 22 bytes ($02a7 for 89 bytes)

        ORG $0801
        DC.B $0b,8,$ef,0        ; '239 SYS2061'
        DC.B $9e,$32,$30,$36
        DC.B $31,0,0,0

        sei             ; disable interrupts
```

```
        inc $d030       ; or "bit $d030" if 2MHz mode is not enabled
        inc 1           ; Select ALL-RAM configuration

        ldx #0          ;** parameter - # of overlap bytes-1 off $ffff
overlap lda $aaaa,x     ;** parameter start of off-end bytes
        sta WRAPBUF,x   ; Copy to wrap/safety buffer
        dex
        bpl overlap

        ldx #block200-end-block200+1    ; $54   ($59 max)
packlp  lda block200-1,x
        sta block200--1,x
        dex
        bne packlp

        ldx #block-stack-end-block-stack+1      ; $b3   (stack! ~$e8 max)
packlp2 lda block-stack-1,x
        dc.b $9d                ; sta $nnnn,x
        dc.w block-stack--1     ; (ZP addressing only addresses ZP!)
        dex
        bne packlp2

        ldy #$aa        ;** parameter SIZE high + 1 (max 255 extra bytes)
cploop  dex             ; ldx #$ff on the first round
        lda $aaaa,x     ;** parameter DATAEND-0x100
        sta $ff00,x     ;** parameter ORIG LEN-0x100+ reserved bytes
        txa             ;cpx #0
        bne cploop
        dec cploop+6
        dec cploop+3
        dey
        bne cploop
        jmp main
```

The first part of the code contains a sys command for the basic interpreter, two loops that copy the decompression code to zeropage/stack ($f7-$1aa) and to the system input buffer ($200-$253). The latter code segment contains byte, bit and Gamma Code input routines and the RLE byte code table, the former code segment contains the rest.

This code also copies the compressed data forward in memory so that it won't be overwritten by the decompressed data before we have had a change to read it. The decompression starts at the beginning and proceeds upwards in both the compressed and decompressed data. A safety area is calculated by the compression routine. It finds out how many bytes we need for temporary data expansion, i.e. for escaped bytes. The wrap buffer is used for files that extend upto the end of memory, and would otherwise overwrite the compressed data with decompressed data before it has been read.

This code fragment is not used during the decompression itself. In fact the code will normally be overwritten when the actual decompression starts.

The very start of the next code block is located inside the zero page and the rest fills the lowest portion of the microprocessor stack. The zero page is used to make the references to different variables shorter and faster. Also, the variables don't take extra code to initialize, because they are copied with the same copy loop as the rest of the code.

```
block-stack
#rorg $f7       ; $f7 - ~$1e0
block-stack-

bitstr  dc.b $80        ; ZP    $80 == Empty
esc     dc.b $00        ; ** parameter (saves a byte when here)

OUTPOS = *+1            ; ZP
putch   sta $aaaa       ; ** parameter
        inc OUTPOS      ; ZP
        bne 0$          ; Note: beq 0$; rts; 0$: inc OUTPOS+1;rts would be
; $0100                 ;       faster, but 1 byte longer
        inc OUTPOS+1    ; ZP
0$      rts
```

putch is the subroutine that is used to output the decompressed bytes. In this case the bytes are written to memory. Because the subroutine call itself takes 12 cycles (6 for jsr and another 6 for rts), and the routine is called a lot of times during the decompression, the routine itself should be as fast as possible. This is achieved by removing the need to save any registers. This is done by using an absolute addressing mode instead of indirect indexed or absolute indexed addressing (sta $aaaa

instead of sta ($zz),y or sta $aa00,y). With indexed addressing you would need to save+clear+restore the index register value in the routine.

Further improvement in code size and execution speed is done by storing the instruction that does the absolute addressing to zero page. When the memory address is incremented we can use zero-page addressing for it too. On the other hand, the most time is spent in the bit input routine so further optimization of this routine is not feasible.

```
newesc  ldy esc         ; remember the old code (top bits for escaped byte)
        ldx #2          ; ** PARAMETER
        jsr getchkf     ; get & save the new escape code
        sta esc
        tya             ; pre-set the bits
        ; Fall through and get the rest of the bits.
noesc   ldx #6          ; ** PARAMETER
        jsr getchkf
        jsr putch       ; output the escaped/normal byte
        ; Fall through and check the escape bits again
main    ldy #0          ; Reset to a defined state
        tya             ; A = 0
        ldx #2          ; ** PARAMETER
        jsr getchkf     ; X=2 -> X=0
        cmp esc
        bne noesc       ; Not the escape code -> get the rest of the byte
        ; Fall through to packed code
```

The decompression code is first entered in main. It first clears the accumulator and the Y register and then gets the escape bits (if any are used) from the input stream. If they don't match with the current escape code, we get more bits to complete a byte and then output the result. If the escape bits match, we have to do further checks to see what to do.

```
        jsr getval      ; X = 0
        sta xstore      ; save the length for a later time
        cmp #1          ; LEN == 2 ?
        bne lz77        ; LEN != 2      -> LZ77
        tya             ; A = 0
        jsr get1bit     ; X = 0
        lsr             ; bit -> C, A = 0
        bcc lz77-2      ; A=0 -> LZPOS+1
        ;***FALL THRU***
```

We first get the Elias Gamma Code value (or actually my independently developed version). If it says the LZ77 match length is greater than 2, it means a LZ77 code and we jump to the proper routine. Remember that the lengths are decremented before encoding, so the code value 1 means the length is 2. If the length is two, we get a bit to decide if we have LZ77 or something else. We have to clear the accumulator, because get1bit does not do that automatically.

If the bit we got (shifted to carry to clear the accumulator) was zero, it is LZ77 with an 8-bit offset. If the bit was one, we get another bit which decides between RLE and an escaped byte. A zero-bit means an escaped byte and the routine that is called also changes the escape bits to a new value. A one-bit means either a short or long RLE.

```
        ; e..e01
        jsr get1bit     ; X = 0
        lsr             ; bit -> C, A = 0
        bcc newesc      ; e..e010
        ;***FALL THRU***

        ; e..e011
srle    iny             ; Y is 1 bigger than MSB loops
        jsr getval      ; Y is 1, get len, X = 0
        sta xstore      ; Save length LSB
        cmp #64         ; ** PARAMETER 63-64 -> C clear, 64-64 -> C set..
        bcc chrcode     ; short RLE, get bytecode

longrle ldx #2          ; ** PARAMETER  111111xxxxxx
        jsr getbits     ; get 3/2/1 more bits to get a full byte, X = 0
        sta xstore      ; Save length LSB

        jsr getval      ; length MSB, X = 0
        tay             ; Y is 1 bigger than MSB loops
```

The short RLE only uses half (or actually 1 value less than a half) of the gamma code range. Larger values switches us into long RLE mode. Because there are several values, we already know some bits of the length value.

Depending on the gamma code maximum value we need to get from one to three bits more to assemble a full byte, which is then used as the less significant part for the run length count. The upper part is encoded using the same gamma code we are using everywhere. This limits the run length to 16 kilobytes for the smallest maximum value (-m5) and to the full 64 kilobytes for the largest value (-m7).

Additional compression for RLE is gained using a table for the 31 top-ranking RLE bytes. We get an index from the input. If it is from 1 to 31, we use it to index the table. If the value is larger, the lower 5 bits of the value gives us the 5 most significant bits of the byte to repeat. In this case we read 3 additional bits to complete the byte.

```
chrcode jsr getval      ; Byte Code, X = 0
        tax             ; this is executed most of the time anyway
        lda table-1,x   ; Saves one jump if done here (loses one txa)

        cpx #32         ; 31-32 -> C clear, 32-32 -> C set..
        bcc 1$          ; 1..31 -> the byte to repeat is in A

        ; Not ranks 1..31, -> 111110xxxxx (32..64), get byte..
        txa             ; get back the value (5 valid bits)
        jsr get3bit     ; get 3 more bits to get a full byte, X = 0

1$      ldx xstore      ; get length LSB
        inx             ; adjust for cpx#$ff;bne -> bne
dorle   jsr putch
        dex
        bne dorle       ; xstore 0..255 -> 1..256
        deym
        bne dorle       ; Y was 1 bigger than wanted originally
mainbeq beq main        ; reverse condition -> jump always
```

After deciding the repeat count and decoding the value to repeat we simply have to output the value enough times. The X register holds the lower part and the Y register holds the upper part of the count. The X register value is first incremented by one to change the code sequence dex ; cpx #$ff ; bne dorle into simply dex ; bne dorle. This may seem strange, but it saves one byte in the decompression code and two clock cycles for each byte that is outputted. It's almost a ten percent improvement. :-)

The next code fragment is the LZ77 decode routine and it is used in the file parts that do not have equal byte runs (and even in some that have). The routine simply gets an offset value and copies a sequence of bytes from the already decompressed portion to the current output position.

```
lz77    jsr getval      ; X=0 -> X=0
        cmp #127        ; ** PARAMETER  Clears carry (is maximum value)
        beq eof         ; EOF

        sbc #0          ; C is clear -> subtract 1  (1..126 -> 0..125)
        ldx #0          ; ** PARAMETER (more bits to get)
        jsr getchkf     ; clears Carry, X=0 -> X=0

lz77-2  sta LZPOS+1     ; offset MSB
        ldx #8
        jsr getbits     ; clears Carry, X=8 -> X=0
                        ; Note: Already eor:ed in the compressor..
        ;eor #255       ; offset LSB 2's complement -1 (i.e. -X = ~X+1)
        adc OUTPOS      ; -offset -1 + curpos (C is clear)
        sta LZPOS

        lda OUTPOS+1
        sbc LZPOS+1     ; takes C into account
        sta LZPOS+1     ; copy X+1 number of chars from LZPOS to OUTPOS
        ;ldy #0         ; Y was 0 originally, we don't change it

        ldx xstore      ; LZLEN
        inx             ; adjust for cpx#$ff;bne -> bne
lzloop  lda (LZPOS),y
        jsr putch
        iny             ; Y does not wrap because X=0..255 and Y initially 0
        dex
        bne lzloop      ; X loops, (256,1..255)
        beq mainbeq     ; jump through another beq (-1 byte, +3 cycles)
```

There are two entry-points to the LZ77 decode routine. The first one (lz77) is for copy lengths bigger than 2. The second entry point (lz77-2) is for the length of 2 (8-bit offset value).

```
        ; EOF
eof     lda #$37        ; ** could be a PARAMETER
        sta 1
        dec $d030       ; or "bit $d030" if 2MHz mode is not enabled
        lda OUTPOS      ; Set the basic prg end address
        sta BASEND
        lda OUTPOS+1
        sta BASEND+1
        cli             ; ** could be a PARAMETER
        jmp $aaaa       ; ** PARAMETER
#rend
block-stack-end
```

Some kind of a end of file marker is necessary for all variable-length codes. Otherwise we could not be certain when to stop decoding. Sometimes the byte count of the original file is used instead, but here a special EOF condition is more convenient. If the high part of a LZ77 offset is the maximum gamma code value, we have reached the end of file and must stop decoding. The end of file code turns on BASIC and KERNEL, turns off 2 MHz mode (for C128) and updates the basic end addresses before allowing interrupts and jumping to the program start address.

The next code fragment is put into the system input buffer. The routines are for getting bits from the encoded message (getbits) and decoding the Elias Gamma Code (getval). The table at the end contains the ranked RLE bytes. The compressor automatically decreases the table size if not all of the values are used.

```
block200
#rorg   $200    ; $200-$258
block200-

getnew  pha             ; 1 Byte/3 cycles
INPOS = *+1
        lda $aaaa       ;** parameter
        rol             ; Shift in C=1 (last bit marker)
        sta bitstr      ; bitstr initial value = $80 == empty
        inc INPOS       ; Does not change C!
        bne 0$
        inc INPOS+1     ; Does not change C!
        bne 0$
        ; This code does not change C!
        lda #WRAPBUF    ; Wrap from $ffff->$0000 -> WRAPBUF
        sta INPOS
0$      pla             ; 1 Byte/4 cycles
        rts


; getval : Gets a 'static huffman coded' value
; ** Scratches X, returns the value in A **
getval  inx             ; X must be 0 when called!
        txa             ; set the top bit (value is 1..255)
0$      asl bitstr
        bne 1$
        jsr getnew
1$      bcc getchk      ; got 0-bit
        inx
        cpx #7          ; ** parameter
        bne 0$
        beq getchk      ; inverse condition -> jump always

; getbits: Gets X bits from the stream
; ** Scratches X, returns the value in A **
get1bit inx             ;2
getbits asl bitstr
        bne 1$
        jsr getnew
1$      rol             ;2
getchk  dex             ;2              more bits to get ?
getchkf bne getbits     ;2/3
        clc             ;2              return carry cleared
        rts             ;6+6


table   dc.b 0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0
        dc.b 0,0,0,0,0,0,0,0
```

```
#rend
block200-end
```

Target Application Compression Tests

The following data compression tests are made on my four C64 test files:
   bs.bin is a demo part, about 50% code and 50% graphics data
   delenn.bin is a BFLI picture with a viewer, a lot of dithering
   sheridan.bin is a BFLI picture with a viewer, dithering, black areas
   ivanova.bin is a BFLI picture with a viewer, dithering, larger black areas

Packer	Size	Left	Comment
bs.bin	41537		
ByteBonker 1.5	27326	65.8%	Mode 4
Cruelcrunch 2.2	27136	65.3%	Mode 1
The AB Cruncher	27020	65.1%	
ByteBoiler (REU)	26745	64.4%	
RLE + ByteBoiler (REU)	26654	64.2%	
PuCrunch	26415	63.6%	-m5
delenn.bin	47105		
The AB Cruncher	N/A	N/A	Crashes
ByteBonker 1.5	21029	44.6%	Mode 3
Cruelcrunch 2.2	20672	43.9%	Mode 1
ByteBoiler (REU)	20371	43.2%	
RLE + ByteBoiler (REU)	19838	42.1%	
PuCrunch	19734	41.9%	-p2
sheridan.bin	47105		
ByteBonker 1.5	13661	29.0%	Mode 3
Cruelcrunch 2.2	13595	28.9%	Mode H
The AB Cruncher	13534	28.7%	
ByteBoiler (REU)	13308	28.3%	
PuCrunch	12526	26.6%	-p2
RLE + ByteBoiler (REU)	12478	26.5%	
ivanova.bin	47105		
ByteBonker 1.5	11016	23.4%	Mode 1
Cruelcrunch 2.2	10883	23.1%	Mode H
The AB Cruncher	10743	22.8%	
ByteBoiler (REU)	10550	22.4%	
PuCrunch	9844	20.9%	-p2
RLE + ByteBoiler (REU)	9813	20.8%	
LhA	9543	20.3%	Decompressor not included
gzip -9	9474	20.1%	Decompressor not included

Calgary Corpus Suite

The original compressor only allows files upto 63 kB. To be able to compare my algorithm to others I modified the compressor to allow bigger files. I then got some reference results using the Calgary Corpus test suite.

Note that the decompression code is included in the compressed files, although it is not valid for files over 63k (compressed or uncompressed size). About 34 bytes are decompression parameters, the rest (approx. 300 bytes) is 6510 machine language. Kolmogorov complexity, anyone ?:-)

To tell you the truth, the results surprised me, because the compression algorithm -IS- developed for a very special case in mind. It only has a fixed code for LZ77/RLE lengths, not even a static one (fixed != static != adaptive)! Also, it does not use arithmetic code (or Huffman) to compress the literal bytes. Because most of the big files are ASCII text, this somewhat handicaps my compressor, although the new tagging system is very

happy with 7-bit ASCII input. Also, decompression is relatively fast, and uses no extra memory.

I'm getting relatively near LhA, and shorter than LhA for 8 files (300-byte decompressor included!), and relatively near or shorter than LhA in other cases if the decompressor is removed.

The table contains the file name (file), compression options (options), the original file size (in) and the compressed file size (out) in bytes, average number of bits used to encode one byte (b/B), remaining size (ratio) and the reduction (gained), and the time used for compression. For comparison, the last three columns show the compressed sizes for LhA, Zip and GZip (with the -9 option), respectively.

FreeBSD epsilon3.vlsi.fi PentiumPro® 200MHz
Estimated decompression on a C64 (1MHz 6510) 6:47

file	options	in	out	b/B	ratio	gained	time	LhA out	Zip out	GZip-9 out
bib	-p4	111261	35457	2.55	31.87%	68.13%	8.3	40740	35041	34900
book1	-p4	768771	318919	3.32	41.49%	58.51%	65.1	339074	313352	312281
book2	-p4	610856	208627	2.74	34.16%	65.84%	43.5	228442	206663	206158
geo	-p2	102400	72812	5.69	71.11%	28.89%	11.4	68574	68471	68414
news	-p3	377109	144566	3.07	38.34%	61.66%	15.2	155084	144817	144400
obj1	-m6	21504	10750	4.00	50.00%	50.00%	0.1	10310	10300	10320
obj2		246814	83046	2.70	33.65%	66.35%	13.5	84981	81608	81087
paper1	-p2	53161	19536	2.94	36.75%	63.25%	1.5	19676	18552	18543
paper2	-p3	82199	30676	2.99	37.32%	62.68%	4.3	32096	29728	29667
paper3	-p2	46526	19234	3.31	41.35%	58.65%	1.4	18949	18072	18074
paper4	-p1 -m5	13286	6095	3.68	45.88%	54.12%	0.2	5558	5511	5534
paper5	-p1 -m5	11954	5494	3.68	45.96%	54.04%	0.1	4990	4970	4995
paper6	-p2	38105	14159	2.98	37.16%	62.84%	0.8	13814	13207	13213
pic	-p1	513216	57835	0.91	11.27%	88.73%	23.2	52221	56420	52381
progc	-p1	39611	14221	2.88	35.91%	64.09%	0.7	13941	13251	13261
progl	-p1	71646	17038	1.91	23.79%	76.21%	3.8	16914	16249	16164
progp		49379	11820	1.92	23.94%	76.06%	1.3	11507	11222	11186
trans	-p2	93695	19511	1.67	20.83%	79.17%	3.7	22578	18961	18862
total		3251493	1089796	2.68	33.52%	66.48%	3:18			

Canterbury Corpus Suite

The following shows the results on the Canterbury corpus. Again, I am quite pleased with the results. For example, pucrunch beats GZip -9 for lcet10.txt if you remove the decompression code.

FreeBSD epsilon3.vlsi.fi PentiumPro® 200MHz
Estimated decompression on a C64 (1MHz 6510) 6:00

file	opt	in	out	b/B	ratio	gained	time	LhA out	Zip out	GZip-9 out
alice29.txt	-p4	152089	55103	2.90	36.24%	63.76%	11.3	59160	54525	54191
ptt5	-p1	513216	57835	0.91	11.27%	88.73%	23.2	52272	56526	52382
fields.c		11150	3505	2.52	31.44%	68.56%	0.1	3180	3230	3136
kennedy.xls		1029744	265887	2.07	25.83%	74.17%	571	198354	206869	209733
sum		38240	13334	2.79	34.87%	65.13%	0.6	14016	13006	12772
lcet10.txt	-p4	426754	144585	2.72	33.89%	66.11%	30.8	159689	144974	144429
plrabn12.txt	-p4	481861	199134	3.31	41.33%	58.67%	43.6	210132	195299	194277
cp.html	-p1	24603	8679	2.83	35.28%	64.72%	0.4	8402	8085	7981
grammar.lsp	-m5	3721	1591	3.43	42.76%	57.24%	0.0	1280	1336	1246
xargs.1	-m5	4227	2117	4.01	50.09%	49.91%	0.0	1790	1842	1756
asyoulik.txt	-p4	125179	50594	3.24	40.42%	59.58%	7.5	52377	49042	48829
total		2810784	802364	2.28	28.55%	71.45%	11:29			

Conclusions

In this article I have presented a compression program which creates compressed executable files for C64, VIC20 and Plus4/C16. The compression can be performed on Amiga, MS-DOS/Win machine or any other machine with a C-compiler. A powerful machine allows asymmetric compression: a lot of resources can be used to compress the data while needing minimal resources for decompression. This was one of the design requirements.

Two original ideas were presented: a new literal byte tagging system and an algorithm using hybrid RLE and LZ77. Also, a detailed explanation of

the LZ77 string match routine and the optima parsing scheme was presented.

The compression ratio and decompression speed is comparable to other compression programs for Commodore 8-bit computers.

But what are then the real advantages of pucrunch compared to traditional C64 compression programs in addition to that you can now compress VIC20 and Plus4/C16 programs? Because I'm lousy at praising my own work, I let you see some actual user comments. I have edited the correspondence a little, but I hope he doesn't mind. My comments are marked with an asterisk. Maybe Steve has something to add also?

---8<----8<----8<----8<----8<----8<----8<----8<----8<----8<----8<---

A big advantage is that pucrunch does RLE and LZ in one pass. For demos I only used a cruncher and did my own RLE routines as it is somewhat annoying to use an external program for this. These programs require some memory and ZP-addresses like the cruncher does. So it can easily happen that the decruncher or depacker interfere with your demo-part, if you didn't know what memory is used by the depacker. At least you have more restrictions to care about. With pucrunch you can do RLE and LZ without having too much of these restrictions.

* Right, and because pucrunch is designed that way from the start, it can
* get better results with one-pass RLE and LZ than doing them separately.
* On the other hand it more or less requires that you _don't_ RLE-pack the
* file first..

This is true, we also found that out. We did a part for our demo which had some tables using only the low-nybble. Also the bitmap had to be filled with a specific pattern. We did some small routines to shorten the part, but as we tried pucrunch, this became obsolete. From 59xxx bytes to 12xxx or 50 blocks, with our own RLE and a different cruncher we got 60 blks! Not bad at all ;)

Not to mention that you have the complete and commented source-code for the decruncher, so that you can easily change it to your own needs. And it's not only very flexible, it is also very powerful. In general pucrunch does a better job than ByteBoiler+Sledgehammer.

In addition to that pucrunch is of course much faster than crunchers on my C64, this has not only to do with my 486/66 and the use of an HDD. See, I use a cross-assembler-system, and with pucrunch I don't have to transfer the assembled code to my 64, crunch it, and transfer it back to my pc. Now, it's just a simple command-line and here we go... And not only I can do this, my friend who has an amiga uses pucrunch as well. This is the first time we use the same cruncher, since I used to take ByteBoiler, but my friend didn't have a REU so he had to try another cruncher.

So, if I try to make a conclusion: It's fast, powerful and extremly flexible (thanks to the source-code).

---8<----8<----8<----8<----8<----8<----8<----8<----8<----8<----8<---

Just for your info...

We won the demo-competition at the Interjam'98 and everything that was crunched ran through the hands of pucrunch... Of course, you have been mentioned in the credits. If you want to take a look, search for KNOOPS/DREAMS, which should be on the ftp-servers in some time.
So, again, THANKS! :)

Ninja/DREAMS

---8<----8<----8<----8<----8<----8<----8<----8<----8<----8<----8<---

So, what can I possibly hope to add to that, right?:-)

If you have any comments, questions, article suggestions or just a general hello brewing in your mind, send me mail or visit my homepage.

See you all again in the next issue!

-Pasi

Appendix: The Log Book

5.3.1997
Tried reverse LZ, i.e. mirrored history buffer. Gained some bytes, but its not really worth it, i.e. the compress time increases hugely and the decompressor gets bigger.

6.3.1997
Tried to have a code to use the last LZ copy position (offset added to the lastly used LZ copy position). On bs.run I gained 57 bytes, but in fact the net gain was only 2 bytes (decompressor becomes ~25 bytes longer, and the lengthening of the long rle codes takes away the rest 30).

10.3.1997
Discovered that my representation of integers 1-63 is in fact an Elias Gamma Code. Tried Fibonacci code instead, but it was much worse (~500 bytes on bs.run, ~300 bytes on delenn.run) without even counting the expansion of the decompression code.

12.3.1997
'huffman' coded RLE byte -> ~70 bytes gain for bs.run. The RLE bytes used are ranked, and top 15 are put into a table, which is indexed by a Elias Gamma Code. Other RLE bytes get a prefix "1111".

15.3.1997
The number of escape bits used is again selectable. Using only one escape bit for delenn.run gains ~150 bytes. If #-option is not selected, automatically selects the number of escape bits (is a bit slow).

16.3.1997
Changed some arrays to short. 17 x inlen + 64kB memory used. opt-escape() only needs two 16-element arrays now and is slightly faster.

31.3.1997
Tried to use BASIC ROM as a codebook, but the results were not so good. For mostly-graphics files there are no long matches -> no net gain, for mostly-code files the file itself gives a better codebook.. Not to mention that using the BASIC ROM as a codebook is not 100% compatible.

1.4.1997
Tried maxlen 128, but it only gained 17 bytes on ivanova.run, and lost ~15 byte on bs.run. This also increased the LZPOS maximum value from ~16k to ~32k, but it also had little effect.

2.4.1997
Changed to coding so that LZ77 has the priority. 2-byte LZ matches are coded in a special way without big loss in efficiency, and codes also RLE/Escape.

5.4.1997
Tried histogram normalization on LZLEN, but it really did not gain much of anything, not even counting the mapping table from index to value that is needed.

11.4.1997
8..14 bit LZPOS base part. Automatic selection. Some more bytes are gained if the proper selection is done before the LZ/RLELEN optimization. However, it can't really be done automatically before that, because it is a recursive process and the original LZ/RLE lengths are lost in the first optimization..

22.4.1997
Found a way to speed up the almost pathological cases by using the RLE table to skip the matching beginnings.

2.5.1997
Switched to maximum length of 128 to get better results on the Calgary Corpus test suite.

25.5.1997
Made the maximum length adjustable. -m5, -m6, and -m7 select 64, 128 and 256 respectively. The decompression code now allows escape bits from 0 to 8.

1.6.1997

Optimized the escape optimization routine. It now takes almost no time at all. It used a whole lot of time on large escape bit values before. The speedup came from a couple of generic data structure optimizations and loop removals by informal deductions.

3.6.1997
Figured out another, better way to speed up the pathological cases. Reduced the run time to a fraction of the original time. All 64k files are compressed under one minute on my 25 MHz 68030. pic from the Calgary Corpus Suite is now compressed in 19 seconds instead of 7 minutes (200 MHz Pentium w/ FreeBSD). Compression of ivanova.run (one of my problem cases) was reduced from about 15 minutes to 47 seconds. The compression of bs.run has been reduced from 28 minutes (the first version) to 24 seconds. An excellent example of how the changes in the algorithm level gives the most impressive speedups.

6.6.1997
Changed the command line switches to use the standard approach.

11.6.1997
Now determines the number of bytes needed for temporary data expansion (i.e. escaped bytes). Warns if there is not enough memory to allow successful decompression on a C64.

Also, now it's possible to decompress the files compressed with the program (must be the same version). (-u)

17.6.1997
Only checks the lengths that are power of two's in OptimizeLength(), because it does not seem to be any (much) worse than checking every length. (Smaller than found maximum lengths are checked because they may result in a shorter file.) This version (compiled with optimizations on) only spends 27 seconds on ivanova.run.

19.6.1997
Removed 4 bytes from the decrunch code (begins to be quite tight now unless some features are removed) and simultaneously removed a not-yet-occurred hidden bug.

23.6.1997
Checked the theoretical gain from using the lastly outputted byte (conditional probabilities) to set the probabilities for normal/LZ77/RLE selection. The number of bits needed to code the selection is from 0.0 to 1.58, but even using arithmetic code to encode it, the original escape system is only 82 bits worse (ivanova.run), 7881/7963 bits total. The former figure is calculated from the entropy, the latter includes LZ77/RLE/escape select bits and actual escapes.

18.7.1997
In LZ77 match we now check if a longer match (further away) really gains more bits. Increase in match length can make the code 2 bits longer. Increase in match offset can make the code even longer (2 bits for each magnitude). Also, if LZPOS low part is longer than 8, the extra bits make the code longer if the length becomes longer than two.

ivanova -5 bytes, sheridan -14, delenn -26, bs -29

When generating the output rescans the LZ77 matches. This is because the optimization can shorten the matches and a shorter match may be found much nearer than the original longer match. Because longer offsets usually use more bits than shorter ones, we get some bits off for each match of this kind. Actually, the rescan should be done in OptimizeLength() to get the most out of it, but it is too much work right now (and would make the optimize even slower).

29.8.1997
4 bytes removed from the decrunch code. I have to thank Tim Rogers (timr@eurodltd.co.uk) for helping with 2 of them.

12.9.1997
Because SuperCPU doesn't work correctly with inc/dec $d030, I made the 2 MHz user-selectable and off by default. (-f)

13.9.1997
Today I found out that most of my fast string matching

algorithm matches the one developed by [Fenwick and Gutmann, 1994]*. It's quite frustrating to see that you are not a genius after all and someone else has had the same idea. :-) However, using the RLE table to help still seems to be an original idea, which helps immensely on the worst cases. I still haven't read their paper on this, so I'll just have to get it and see..

* [Fenwick and Gutmann, 1994]. P.M. Fenwick and P.C. Gutmann, "Fast LZ77 String Matching", Dept of Computer Science, The University of Auckland, Tech Report 102, Sep 1994

14.9.1997
The new decompression code can decompress files from $258 to $ffff (or actually all the way upto $1002d :-). The drawback is: the decompression code became 17 bytes longer. However, the old decompression code is used if the wrap option is not needed.

16.9.1997
The backSkip table can now be fixed size (64 kWord) instead of growing enormous for "BIG" files. Unfortunately, if the fixed-size table is used, the LZ77 rescan is impractical (well, just a little slow, as we would need to recreate the backSkip table again). On the other hand the rescan did not gain so many bytes in the first place (percentage). The define BACKSKIP-FULL enables the old behavior (default). Note also, that for smaller files than 64kB (the primary target files) the default consumes less memory.

The hash value compare that is used to discard impossible matches does not help much. Although it halves the number of strings to consider (compared to a direct one-byte compare), speedwise the difference is negligible. I suppose a mismatch is found very quickly when the strings are compared starting from the third charater (the two first characters are equal, because we have a full hash table). According to one test file, on average 3.8 byte-compares are done for each potential match. A define HASH-COMPARE enables (default) the hash version of the compare, in which case "inlen" bytes more memory is used.

After removing the hash compare my algorithm quite closely follows the [Fenwick and Gutmann, 1994] fast string matching algorithm (except the RLE trick). (Although I *still* haven't read it.)

14 x inlen + 256 kB of memory is used (with no HASH-COMPARE and without BACKSKIP-FULL).

18.9.1997
One byte removed from the decompression code (both versions).

30.12.1997
Only records longer matches if they compress better than shorter ones. I.e. a match of length N at offset L can be better than a match of length N+1 at 4*L. The old comparison was "better or equal" (">="). The new comparison "better" (">") gives better results on all Calgary Corpus files except "geo", which loses 101 bytes (0.14% of the compressed size).

An extra check/rescan for 2-byte matches in OptimizeLength() increased the compression ratio for "geo" considerably, back to the original and better. It seems to help for the other files also. Unfortunately this only works with the full backskip table (BACKSKIP-FULL defined).

21.2.1998
Compression/Decompression for VIC20 and C16/+4 incorporated into the same program.

16.3.1998
Removed two bytes from the decompression codes.

17.8.1998
There was a small bug in pucrunch which caused the location $2c30 to be decremented (dec $2c30 instead of bit $d030) when run without the -f option. The source is fixed and executables are now updated.

References

1. http://www.cs.tut.fi/~albert/
2. http://www.cs.tut.fi/~albert/Dev/
2. http://www.cs.tut.fi/~albert/Dev/pucrunch/

....
....
..
. C=H #17

::

VIC-20 Kernal ROM Disassembly Project
Richard Cini

Introduction

In order to put this project into perspective, a little personal history is needed. I received my first Commodore as a gift from my parents back in 1982. I used Commodore PETs in the school's computer lab, and Radio Shack Model I's in the local R/S store. It was nice to have one of my own to hack on, though. Back then, most of my work was with the built-in BASIC interpreter. My claim to fame (to my family, at least) was a BASIC/machine language mailing list management program, and an allophone speech synthesizer hardware-software hack. Both worked well, which surprised my mother, who claims that nothing I ever built worked right. As I grew-up, my computer-of-choice changed, but I never lost my love for the VIC. It is small, easy to program, has a very capable processor (by 1980's standards) and decent I/O capability. It's peripherals were varied, if not quirky (take the 1515 printer, for example), but at least everything worked well together.

Now, fast forward to 1994. Commodore International failed, crippled by years of a weak product strategy, squandered opportunities and the market's increased focus on mainstream PC-compatible or Macintosh machines as productivity tools. After Commodore's failure, I decided that I wanted to try to purchase the Commodore 8-bit intellectual property, including the rights to the Kernal and BASIC source code, primarily for preservation purposes. One of my hobbies is collecting and preserving obsolete and unsupported computers, accessories, documentation, etc.

Since Commodore's bankruptcy attorney would not return my calls (no surprise there), I embarked on decompiling the Kernal. This project, although time consuming and rewarding from an informational perspective was not truly trail-blazing. Many before me probably decompiled parts of the Kernal in order to gain some understanding as it related to another project. However, in my research, I don't recall ever seeing complete recompileable source code. Memory and ROM maps, yes; source code, no.

Marko Makela manages a great Commodore web site that contains lots of useful information, including these memory and ROM maps. These provided the starting point for my work. See http://www.hut.fi/misc/cbm/docs/ for this and much more information. What I would like to accomplish in a series of articles is to explain the process and to discuss specific Kernal routines that may be of interest to C=Hacking readers. Also, where appropriate, I will make comparisons with the other Commodore machines that were the contemporaries of the VIC, the C64 and the PET, specifically.

One might ask, "Why is this project different from all of the other resources already available?" In short, here's why:

1. The end result is a fully modifiable and compilable source file.
2. Only the VIC memory map and ROM location map are available *because* to date everyone has focused on the C64 as it is the more functional (and hence, more popular) Commodore of the era. The same goes for the PET, too.
3. A far as I know, there was no "Mapping the VIC" written, because of #2, above.
4. Because of #3, the fact that I had some spare time, and that I love my VIC (although I don't use it as much these days), I felt that I had to do it.

Brief VIC-20 History

Although I'd like to provide a complete history lesson on the VIC-20, many others before me have done a better job. The March 1985 issue of the IEEE Spectrum has an article, as does Marko's web site. See http://www.hut.fi/misc/cbm/docs/peddle.english.html for a great article on Chuck Peddle, the well-known creator of the 6502 microprocessor. Nonetheless, I will provide a Readers' Digest version of the history of the VIC.

In the late-70s, engineers in the Advanced Systems Design Group (ASDG) at Commodore created a multi-function video/sound interface chip, the 6560 (a.k.a., the VIC). Al Charpentier ran the LSI section of the ASDG, and was the lead in designing the VIC-I, and later, the VIC-II. The ASDG was the old MOS Semiconductor operation that Commodore bought in the mid-70s. The 6560 supported complete composite color video, 3-voice plus white noise

sound, a volume control, two analog-to-digital converters that supported the use of a game paddle or joystick, and a light pen interface. Feature-rich as it was, no manufacturer wanted to commit a product line to it. Since Commodore could not find anyone to buy the chip, they decided to build a computer that featured the chip. Hence, the VIC-20 was born. Al's buddy, Bob Yannes, was a senior systems designer at Commodore who developed the VIC-20 (and later, the C64) prototype. The VIC was Commodore's first color computer, and the first designed for home use.

Relationships with other Commodore Products

The structure of the VIC-20 ROM parallels those of Commodore's other machines of that era, the PET and the C64. All three machines share the concept of a standard, public, API accessed through a jump table located in the last page of the system ROM.

There's also another striking similarity, which comes somewhat as a surprise to this writer, but intuitively makes sense. The C64 and VIC Kernal ROMs are nearly identically laid-out and contain a lot of common code. The C64 ROM of course contains certain modifications relating to its unique hardware and capabilities, but otherwise is the same. Since the machines are so similar in design (the same engineer designed them), the similarity of the code isn't surprising. This recycling appears to have been a cost-effective way to develop a new computer in record time.

The Process

I believe that the reverse engineering process for the purpose of creating source code that produces binary-identical object code, is fairly standard: know thy hardware, get the object code, turn object code into assembly code, give everything names, test compile, fix errors, re-compile and call it done. However, I'm certain that C=Hacking readers have reverse engineering methods which differ from mine. I'd be interested in hearing them, as I'm always looking for a better way to do things. I started with an image of the ROM from my VIC (although a ROM image from funet would work) and cranked it through a disassembler (I used SuperMon). Since I wasn't too up on transferring data from the VIC to the PC, I used a brute-force method: scanning disassembler listings into TIFF files and running them through an OCR program. This produced plain-text files for me to work with. Then, I used various available information on the Web and in books (such as "VIC Revealed", "The Commodore Innerspace Anthology", "Mapping the 64") to break the code into subroutines. I inserted meaningful memory and program location labels, taking names from all of the above sources. My assembler allowed me to create conventional data and code "segments", so I took the time to create a "real" data segment that mirrored the VIC memory map. These segments are not to be confused with the segments supported under the various PeeCee memory models. I used TASM 3.1, a shareware 8-bit table-based assembler by Squak Valley Software of Issaquah, WA. TASM supports the 6502, 6800/05/11, 8048/51/85/96, Z80 and TMS7000/32010/32025 processors.

Next, I read the code and added comments as I went along. I do this on a routine-by-routine basis, as time permits, but always beginning at the first instruction after POR. Finally, I did a recompilation and compared the recompiled output with the ROM image to make sure that no errors were introduced in the source code creation process. I wasn't so lucky the first time around :-). I like to make gross checks first: ROM image size, location of well-known routines (such as the jump table at $FF85, the POR vector at $FD22, and the individual jump table routines). This helps to narrow down the location of any errors.

Once I was satisfied that the disassembly was right, I created ROM image to be burned into a test EPROM. The ROMs used in the VIC are 2364 mask-programmed ROMs. 2364s are a 24-pin 8k x 8bit ROM device, and the closest commonly available EPROM is the 2764 EPROM, a 28-pin 8k x 8bit device. Since the 2364s have four fewer pins (reflecting its non-programmability), an adapter board needs to be built. A piece of perf board and two 28-pin wire-wrap DIP socket should do the trick.

If all checks-out, the source is then ready for modification.

Issues and Considerations

There is one important thing that I discovered while hacking the Kernal -- there is no wasted space in the Kernal ROM as it presently exists. Actually, the Kernal occupies 1,279 bytes less than 8k, beginning at $E500 (and consequently, the BASIC ROM overhangs the $E000 boundary by 1,279 bytes). The Kernal developers maximized the 8k space, so any Kernal hack will have to use jumps to a patch area elsewhere in the processor address space. For example, let's say that I wanted to add BASIC 4.0 disk commands to the VIC Kernal by hacking it (recognizing that I could have used the easier wedge method). I could place jumps in the Kernal ROM to locations within my own non-autostart ROM located at $A000. The other

important consideration is backward compatibility. Certain programs may rely on the specific location of code within the Kernal ROM. For example, a game ROM may make direct calls to the internal Plot routine, as opposed to using the jump table at the end of the Kernal ROM (saving a few processor ticks in the process). Shifting code around would relocate that code, breaking that program.

General Hardware Information

The Microprocessor

Manufactured by MOS and second-sourced from Rockwell Semiconductors, the stock 6502 is an 8-bit, 1MHz NMOS-process processor. It supports 56 instructions in 13 addressing modes (although six are combinations of the seven basic modes), three processor registers (.A, .X, and .Y), stack pointer, and a condition code (flags) register. The 6502 supports both maskable and non-maskable interrupts with fixed vectors at the top of its 64k address space.

I/O Processors

VIC-20 I/O is managed by two memory-mapped I/O processors, the 6522 versatile interface adapter (VIA). The I/O region occupies the 4k-address space beginning at $9000. The VIAs manage the keyboard, joystick, light pen, cassette deck, the IEEE serial interface and the user port. Each 6522 has 16 bi-directional I/O lines, four handshaking lines, two 8-bit shift registers and two clock generators (capable of generating free-running or triggered pulses). One of these clocks is responsible for RTC, RS-232, IEEE, and cassette tape timing. Of the eight handshaking lines, two are free for general use, while the other six are used for the IEEE serial port, the RESTORE key, and cassette control. 24-bits of the total 32-bits of I/O are used for keyboard scanning and joystick, light pen, and serial inputs. Truly available to the user is four handshaking lines and 8-bits of I/O.

Video Interface

The VIC-20 video interface is managed by the 6560/6561 VIC chip. The VIC screen is organized in 22 columns by 24 rows in text mode and 176 by 192 pixels in "graphics" mode. Graphics mode is synthesized by mapping the character generator ROM to RAM and modifying the character glyphs. The on-chip video sync generator is capable of generating video in NTSC (6560) or PAL (6561) formats in 16 colors. The VIC also contains three programmable tone generators, a white noise source, volume control, two A/D converters (for game paddle interfacing), a light pen input, screen centering, and independent control over background, foreground, and border colors. The 6560/6561 performs its own DMA to separate 4-bit video RAM and 8-bit character generator ROM. The address buss is a private 2MHz buss, which is not shared with the 1MHz microprocessor buss. Shared ROM access is performed during the processor Phase 1 clock.

Memory Map

Space limitations prohibit listing the complete VIC-20 memory map, but an abridged version may be helpful:

HEX Offset	DESCRIPTION
0000-00FF	Zero page: Kernal and BASIC system areas
0100-01FF	Page 1: tape error log area, processor stack
0200-02FF	Page 2: BASIC input buffer, file and device address tables, keyboard and screen vars, RS232 vars
0300-03FF	Page 3: BASIC vectors, processor register storage, Kernal vectors cassette buffer area
0400-0FFF	Pages 4-15: 3k expansion area
1000-1DFF	User Basic area (unexpanded VIC)
1E00-1FFF	Screen memory (unexpanded VIC)
2000-3FFF	8K expansion RAM/ROM block 1
4000-5FFF	8K expansion RAM/ROM block 2
6000-7FFF	8K expansion RAM/ROM block 3

NOTE: When additional memory is added to block 1, 2 or 3, the Kernal relocates the following things for BASIC:

1000-11FF	Screen memory
1200-?	User Basic area
9400-95FF	Color RAM
8000-8FFF	4K Character generator ROM
8000-83FF	Upper case and graphics
8400-87FF	Reversed upper case and graphics
8800-8BFF	Upper and lower case
8C00-8FFF	Reversed upper and lower case
9000-93FF	I/O Block 0
9000-900F	VIC chip registers
9110-911F	6522 VIA#1 registers
9120-912F	6522 VIA#2 registers
9400-95FF	location of COLOR RAM with additional RAM at blk 1

```
9600-97FF       Normal location of COLOR RAM
9800-9BFF       I/O Block 2
9C00-9FFF       I/O Block 3
A000-BFFF       8K block for expansion ROM (autostart ROM)
C000-DFFF       8K BASIC ROM
E000-FFFF       8K Kernal ROM
```

Kernal Functions

System Startup

Let's first take a look at how a VIC-20 boots. The process is substantially similar for the C64 and the PET (through the commonality of the microprocessor upon which each machine is based), although the locations of various routines differs, as does the memory and I/O map. When power is first applied to the microprocessor, the RESET pin is held low by a 555 timer for a period long enough for the power supply and clock generator to stabilize. The 6502 utilizes the last six bytes of the address space to store three critical vectors: the NMI, RESET, and IRQ vectors, respectively. On power-up, the PC (program counter) is initialized to the address stored at location $FFFC and execution begins at that location (the first column in the source code represents the program line number):

```
6495   FFFA ;=================================================
6496   FFFA ; - Power-on and hardware vectors
6497   FFFA ;
6498   FFFA A9 FE               .dw NMI         ;non-maskable interrupt
6499   FFFC 22 FD               .dw RESET       ;POR
6500   FFFE 72 FF               .dw IRQ         ;IRQ processor
```

Execution begins at $FD22, the POR (power-on reset) vector:

```
5971   FD22  ;#################################################
5972   FD22  ; Power-on RESET entry
5973   FD22  ;#################################################
5974   FD22              RESET
5975   FD22 A2 FF                LDX #$FF
5976   FD24 78                   SEI             ;kill interrupts
5977   FD25 9A                   TXS             ;set stack top
5978   FD26 D8                   CLD
5979   FD27 20 3F FD             JSR SCNROM      ;check for autostart ROM
5980   FD2A D0 03                BNE SKIPA0      ;not there, skip ROM init
5981   FD2C
5982   FD2C 6C 00 A0             JMP (A0BASE)    ;jump to ROM init if present
5983   FD2F
5984   FD2F              SKIPA0
5985   FD2F 20 8D FD             JSR RAMTAS      ;test RAM
5986   FD32 20 52 FD             JSR IRESTR      ;init work memory
5987   FD35 20 F9 FD             JSR IOINIT      ;setup hardware
5988   FD38 20 18 E5             JSR CINT1       ;init video
5989   FD3B 58                   CLI             ;re-enable interrupts
5990   FD3C 6C 00 C0             JMP (BENTER)    ;enter BASIC
```

The startup routines setup the processor stack, check for the existence of an autostart ROM. Autostart ROMs are located in the $A000 block and have a five-byte signature (A0CBM) at offset $04. If the signature is found, the Kernal jumps to the A0ROM initialization routine pointed to by offset $00 of the autostart ROM.

If no signature is found, the Kernal initialization continues by testing the RAM, initializing the system variables, system hardware, and the screen. Finally, the Kernal transfers control to the BASIC initialization entry point at $C000.

Routine SCNROM

The first routine called from the POR code is the SCNROM routine. This routine looks for the special 5-byte signature that indicates the presence of an autostart ROM located in the $A segment.

```
5992   FD3F  ;=================================================
5993   FD3F  ; SCNROM - Scan ROM areas for Autostart ROM signature
5994   FD3F  ;
5995   FD3F              SCNROM
5996   FD3F A2 05                LDX #$05        ;5 chars to compare
5997   FD41
5998   FD41              SCNLOOP
5999   FD41 BD 4C FD             LDA SCANEX,X    ;start at end of signature
6000   FD44 DD 03 A0             CMP A0BASE+3,X  ;compare to ROM sig area
6001   FD47 D0 03                BNE SCANEX      ;no match, exit loop
6002   FD49
6003   FD49 CA                   DEX             ;match; check next char
6004   FD4A D0 F5                BNE SCNLOOP     ;loop
```

```
6005   FD4C
6006   FD4C              SCANEX
6007   FD4C 60                   RTS             ;return; Z=0 if no match
6008   FD4D              ;
6009   FD4D              ; ROMSIG - Autostart ROM signature
6010   FD4D              ;
6011   FD4D              ROMSIG
6012   FD4D 4130C3C2CD           .db "A0", $C3, $C2, $CD  ;A0CBM
```

Routine RAMTAS

The RAMTAS routine is the second subroutine in the initialization process. It clears the first three pages of RAM, then searches for expansion memory. If any is found, the screen memory, color memory, and start of BASIC RAM pointers are adjusted to their documented alternates.

```
6058   FD8D  ;=================================================
6059   FD8D  ; RAMTAS - Initialize system contents
6060   FD8D  ;
6061   FD8D              RAMTAS
6062   FD8D A9 00                LDA #$00        ;zero regs .A and .X
6063   FD8F AA                   TAX
6064   FD90
6065   FD90              RAMTSLP1                ;clear system memory areas
6066   FD90 95 00                STA USRPOK,X    ;zero page
6067   FD92 9D 00 02             STA BUF,X       ;clear page 2
6068   FD95 9D 00 03             STA ERRVPT,X    ;clear page 3
6069   FD98 E8                   INX
6070   FD99 D0 F5                BNE RAMTSLP1    ;loop till done
6071   FD9B
6072   FD9B A2 3C                LDX #$3C        ;setup cassette buffer
6073   FD9D A0 03                LDY #$03        ;area to $033c
6074   FD9F 86 B2                STX TAPE1
6075   FDA1 84 B3                STY TAPE1+1
6076   FDA3 85 C1                STA STAL        ;clear I/O start address…
6077   FDA5 85 97                STA REGSAV      ;…register save
6078   FDA7 8D 81 02             STA OSSTAR      ;…and start of OS memory ptr
6079   FDAA A8                   TAY             ; .Y=0
6080   FDAB A9 04                LDA #$04        ;check RAM from $0400
6081   FDAD 85 C2                STA STAL+1      ;set I/O start to page 3
6082   FDAF
6083   FDAF              RAMTASLP2
6084   FDAF E6 C1                INC STAL        ;increment LSB
6085   FDB1 D0 02                BNE RAMTAS1     ;not done with page, cont.
6086   FDB3
6087   FDB3 E6 C2                INC STAL+1      ;inc. to new page
6088   FDB5
6089   FDB5              RAMTAS1
6090   FDB5 20 91 FE             JSR MEMTST      ;test RAM
6091   FDB8 A5 97                LDA REGSAV
6092   FDBA F0 22                BEQ RAMTAS3
6093   FDBC B0 F1                BCS RAMTASLP2   ;next address
6094   FDBE
6095   FDBE A4 C2                LDY STAL+1      ;done testing,get RAM top MSB…
6096   FDC0 A6 C1                LDX STAL        ;…and LSB
6097   FDC2 C0 20                CPY #$20        ; top at $2000
6098   FDC4 90 25                BCC I6561LP     ;page below $2000, halt
6099   FDC6
6100   FDC6 C0 21                CPY #$21        ;RAM at $2000?
6101   FDC8 B0 08                BCS RAMTAS2     ;yes, set params
6102   FDCA
6103   FDCA A0 1E                LDY #$1E        ;$1E00
6104   FDCC 8C 88 02             STY HIPAGE
6105   FDCF
6106   FDCF              RAMTAS1A
6107   FDCF 4C 7B FE             JMP STOTOP      ;CLC and set RAM top
6108   FDD2
6109   FDD2              RAMTAS2
6110   FDD2 A9 12                LDA #$12        ;With exp. RAM, BASIC starts
6111   FDD4 8D 82 02             STA OSSTAR+1    ;at $1200…
6112   FDD7 A9 10                LDA #$10        ;…and screen starts at $1000
6113   FDD9 8D 88 02             STA HIPAGE
6114   FDDC D0 F1                BNE RAMTAS1A    ;set top of RAM and exit
6115   FDDE
6116   FDDE              RAMTAS3
6117   FDDE 90 CF                BCC RAMTASLP2   ;loop to next address
6118   FDE0
6119   FDE0 A5 C2                LDA STAL+1      ;get MSB of I/O start
6120   FDE2 8D 82 02             STA OSSTAR+1    ;save as start of OS
6121   FDE5 85 97                STA REGSAV      ;save copy
6122   FDE7 C9 11                CMP #$11        ;page $11
6123   FDE9
```

```
6124   FDE9             RATS3
6125   FDE9 90 C4               BCC RAMTASLP2
6126   FDEB
6127   FDEB             I6561LP
6128   FDEB 20 C3 E5            JSR V6561I-2    ;$E5C3 init VIC regs
6129   FDEE 4C EB FD            JMP I6561LP
```

This routine actually tests the RAM, and is called during the memory search loop at $FDB5. It uses a simple walking-bit pattern to test for memory defects:

```
6271   FE91  ;=================================================
6272   FE91  ; MEMTST - Test memory
6273   FE91  ;          .Y is index in page
6274   FE91             MEMTST
6275   FE91 B1 C1               LDA (STAL),Y    ;get address
6276   FE93 AA                  TAX             ;save .A
6277   FE94 A9 55               LDA #%01010101  ;set pattern
6278   FE96 91 C1               STA (STAL),Y    ;save it…
6279   FE98 D1 C1               CMP (STAL),Y    ;…and compare
6280   FE9A D0 08               BNE MEMTS1      ;not equal
6281   FE9C                                     ;pattern compares OK
6282   FE9C 6A                  ROR A           ;%10101010 invert pattern
6283   FE9D 91 C1               STA (STAL),Y    ;save it…
6284   FE9F D1 C1               CMP (STAL),Y    ;…and compare
6285   FEA1 D0 01               BNE MEMTS1      ;not equal
6286   FEA3 A9                  .db $A9         ;LDA #$18 for OK, $55 or $AA
6287   FEA4                                     ; for failed pattern
6288   FEA4             MEMTS1
6289   FEA4 18                  CLC             ;CLC only on error
6290   FEA5 8A                  TXA             ;restore previous .A
6291   FEA6 91 C1               STA (STAL), Y   ;save it
6292   FEA8 60                  RTS
```

The RAMTAS routine also calls STOTOP to save the top of RAM pointer:

```
6241   FE73  ;=================================================
6242   FE73  ; IMEMTP - Set/read top of memory (internal)
6243   FE73  ;  On entry, SEC to read, .X/.Y is LSB/MSB
6244   FE73  ;            CLC to set, .X/.Y is LSB/MSB
6245   FE73  ;
6246   FE73             IMEMTP
6247   FE73 90 06               BCC STOTOP      ;set or read?
6248   FE75 AE 83 02            LDX OSTOP       ;get top of memory
6249   FE78 AC 84 02            LDY OSTOP+1
6250   FE7B
6251   FE7B             STOTOP
6252   FE7B 8E 83 02            STX OSTOP       ;set top of memory
6253   FE7E 8C 84 02            STY OSTOP+1
6254   FE81 60                  RTS
```

Routine IRESTR

This routine loads (or re-loads) the default Kernal vectors upon POR or Run-Stop/Restore sequences. The default vectors that are loaded include the links to IRQ, NMI, Open, Close, Channel In, Channel Out, Clear Channels, Character In, Character Out, Scan Stop Key, Get Keyboard Character, Close All, Load and Save routines within the Kernal ROM.

```
6014   FD52  ;=================================================
6015   FD52  ; IRESTR - Restore KERNAL hardware vectors (internal)
6016   FD52  ;          Called during POR and NMI sequences.
6017   FD52  ;
6018   FD52             IRESTR
6019   FD52 A2 EA               LDX #$EA        ;FIXUP2;#$6D points to list of
6020   FD54 A0 EA               LDY #$EA        ;FIXUP2+1;#$FD $FD6D KERNAL vecs
6021   FD56 18                  CLC
6022   FD57             ;
6023   FD57             ; IVECTR - Change vectors for user
6024   FD57             ;       On entry, SEC= read vector to .X/.Y LSB/MSB
6025   FD57             ;                 CLC= set vector from .X/.Y LSB/MSB
6026   FD57             ;
6027   FD57             IVECTR
6028   FD57 86 C3               STX MEMUSS      ;save vector to temp
6029   FD59 84 C4               STY MEMUSS+1
6030   FD5B A0 1F               LDY #$1F        ;# of bytes to move
6031   FD5D
6032   FD5D             VECLOOP
6033   FD5D B9 B6 02            LDA IRQVP,Y     ;get old vector address
6034   FD60 B0 02               BCS VECSK       ;branch on CY=1/read
6035   FD62
6036   FD62 B1 C3               LDA (MEMUSS),Y  ;get new vector address
```

```
6037   FD64
6038   FD64              VECSK
6039   FD64 91 C3                STA (MEMUSS),Y  ;save new address to temp
6040   FD66 99 B6 02             STA IRQVP,Y     ;and to vector area
6041   FD69 88                   DEY             ;go to next one
6042   FD6A 10 F1                BPL VECLOOP     ;loop
6043   FD6C 60                   RTS
6044   FD6D
6045   FD6D              ;
6046   FD6D              ;KERNAL Vectors
6047   FD6D              ;
6048   FD6D              KNRLSV
6053   FD6D BFEAD2FEADFE         .dw IRQVEC, WARMST, LNKNMI, IOPEN
6053   FD73 0AF4
6054   FD75 4AF3C7F209F3         .dw ICLOSE, ICHKIN, ICHKOT, ICLRCH
6054   FD7B F3F3
6055   FD7D 0EF27AF270F7         .dw ICHRIN, ICHROT, ISTOP,  IGETIN
6055   FD83 F5F1
6056   FD85 EFF3D2FE49F5         .dw ICLALL, WARMST, LNKLOD, LNKSAV
6056   FD8B 85F6
```

Routine IOINIT

This routine initializes the VIAs. Lots of bit twiddling goes on here to set-up the various ports. This routine also starts the system IRQ timer.

```
6138   FDF9  ;=================================================
6139   FDF9  ; IOINIT - Initialize I/O registers
6140   FDF9  ;
6141   FDF9              IOINIT
6142   FDF9 A9 7F                LDA #%01111111  ;disable HW interrupts
6143   FDFB 8D 1E 91             STA D1IER       ;interrupt enable reg VIA1
6144   FDFE 8D 2E 91             STA D2IER       ;interrupt enable reg VIA2

6145   FE01 A9 40                LDA #%01000000  ;Sets tmr1/VIA2 to free-
                                                 ;running; used for IRQ
6146   FE03 8D 2B 91             STA D2ACR       ;VIA2 aux ctrl reg

6147   FE06 A9 40                LDA #%01000000  ;same for tmr1/VIA1. Used for
                                                 ; RS-232 timing
6148   FE08 8D 1B 91             STA D1ACR       ;VIA1 aux ctrl reg

6149   FE0B A9 FE                LDA #%11111110  ;sets CA1/2, CB1/2 modes
                                                 ; CA2/CB2 manual H, CB1 pos
                                                 ; trig. CA1 negative trig.
                                                 ;CA1=Restore
                                                 ;CA2=cassette motor
                                                 ;CB1=user port
                                                 ;CB2=user port
6150   FE0D 8D 1C 91             STA D1PCR       ;VIA1 periph ctrl reg

6151   FE10 A9 DE                LDA #%11011110  ;sets CA1/2, CB1/2 modes
                                                 ; CB2 manual L, CB1 pos
                                                 ; trig. CA1 negative trig.
                                                 ; CA2 manual H
                                                 ;CA1=cassette read
                                                 ;CA2=*SCLK
                                                 ;CB1=*SRQIN
                                                 ;CB2=*SDATAOUT
6152   FE12 8D 2C 91             STA D2PCR       ;VIA2 periph ctrl reg

6153   FE15 A2 00                LDX #$00        ;DDR all bits IN
6154   FE17 8E 12 91             STX D1DDRB      ;VIA1/B data dir reg

6155   FE1A A2 FF                LDX #%11111111  ;DDR all bits OUT
6156   FE1C 8E 22 91             STX D2DDRB      ;VIA2/B data dir reg

6157   FE1F A2 00                LDX #$00        ;DDR all bits IN
6158   FE21 8E 23 91             STX D2DDRA      ;VIA2/A data dir reg

6159   FE24 A2 80                LDX #%10000000  ;BIT7=OUT, BITS6-0 IN
6160   FE26 8E 13 91             STX D1DDRA      ;VIA1/A data dir reg

6161   FE29 A2 00                LDX #$00
6162   FE2B 8E 1F 91             STX D1ORAH      ;VIA1 output reg A MSB

6163   FE2E 20 84 EF             JSR SCLK1       ;set IEEE clock line=1

6164   FE31 A9 82                LDA #%10000010  ;enable IER CA1/VIA1 RESTOR
6165   FE33 8D 1E 91             STA D1IER       ;VIA1 IER BIT7=1
```

```
6166   FE36 20 8D EF            JSR SCLK0       ;set IEEE clock line=0
6167   FE39                ;
6168   FE39                ; ENABTM - Enable timers
6169   FE39                ;
6170   FE39                ENABTM
6171   FE39 A9 C0               LDA #%11000000  ;enable tmr1/VIA2 (IRQ)
6172   FE3B 8D 2E 91            STA D2IER       ;VIA2 IER BIT7-6=1

6173   FE3E A9 89               LDA #%10001001  ;$89 IRQ tic divisor LSB
6174   FE40 8D 24 91            STA D2TM1L      ;VIA2 tmr1 LSB

6175   FE43 A9 42               LDA #%01000010  ;$42 IRQ tic divisor MSB
6176   FE45 8D 25 91            STA D2TM1L+1    ;VIA2 tmr1 MSB
6177   FE48 60                  RTS
```

Routine CINT1

This final routine initializes the character generator, sets the initial screen colors, clears the screen and "homes" the cursor, and updates screen and cursor pointers.

```
1349   E518  ;=================================================
1350   E518  ; CINT1 - Initialize I/O
1351   E518  ;
1352   E518
1353   E518                ;
1354   E518                ;Screen reset
1355   E518                ;
1356   E518                CINT1
1357   E518 20 BB E5            JSR IODEF1      ;set deflt I/O and init VIC
1358   E51B AD 88 02            LDA HIPAGE      ;get screen memory page
1359   E51E 29 FD               AND #%11111101  ;$FD MS nibble is ChrROM and
1360   E520 0A                  ASL A           ;LS nibble is ChrRAM
1361   E521 0A                  ASL A
1362   E522 09 80               ORA #%10000000  ;$80
1363   E524 8D 05 90            STA VRSTRT      ;set chargen ROM to $8000
1364   E527 AD 88 02            LDA HIPAGE      ;get screen mem page
1365   E52A 29 02               AND #%00000010  ;$02 check for screen RAM at
1366   E52C F0 08               BEQ CINT1A      ;$E536 $1E page
1367   E52E
1368   E52E A9 80               LDA #%10000000  ;$80 screen RAM is at $10 page
1369   E530 0D 02 90            ORA VRCOLS      ;set Bit7
1370   E533 8D 02 90            STA VRCOLS
1371   E536
1372   E536                CINT1A
1373   E536 A9 00               LDA #$00
1374   E538 8D 91 02            STA SHMODE      ;enable shift-C=
1375   E53B 85 CF               STA BLNON       ;start at no blink
1376   E53D
1377   E53D A9 EA               LDA #$EA        ;FIXUP1+34;#$DC
1378   E53F 8D 8F 02            STA FCEVAL
1379   E542 A9 EA               LDA #$EA        ;FIXUP1+35;#$EB
1380   E544 8D 90 02            STA FCEVAL+1    ;shift mode evaluation
1381   E547
1382   E547 A9 0A               LDA #$0A
1383   E549 8D 89 02            STA KBMAXL      ;key buffer=16
1384   E54C 8D 8C 02            STA KRPTDL      ;repeat delay=16ms
1385   E54F A9 06               LDA #$06
1386   E551 8D 86 02            STA CLCODE      ;color=6(blue)
1387   E554 A9 04               LDA #$04
1388   E556 8D 8B 02            STA KRPTSP      ;repeat speed
1389   E559 A9 0C               LDA #$0C
1390   E55B 85 CD               STA BLNCT       ;blink timer=12ms
1391   E55D 85 CC               STA BLNSW       ;set for solid cursor
1392   E55F                ;
1393   E55F                ; Clear screen
1394   E55F                ;
1395   E55F                CLRSCN
1396   E55F AD 88 02            LDA HIPAGE      ;mem page for screen RAM
1397   E562 09 80               ORA #%10000000  ;$80
1398   E564 A8                  TAY
1399   E565 A9 00               LDA #$00
1400   E567 AA                  TAX
1401   E568
1402   E568                CLRLP1
1403   E568 94 D9               STY SLLTBL,X    ;address of screen line
1404   E56A 18                  CLC
1405   E56B 69 16               ADC #$16        ;add 22
1406   E56D 90 01               BCC CLRSC1
1407   E56F
1408   E56F C8                  INY
1409   E570
```

```
1410   E570             CLRSC1
1411   E570 E8                  INX
1412   E571 E0 18               CPX #$18        ;all rows done?
1413   E573 D0 F3               BNE CLRLP1
1414   E575
1415   E575 A9 FF               LDA #$FF
1416   E577 95 D9               STA SLLTBL,X
1417   E579 A2 16               LDX #$16
1418   E57B
1419   E57B             CLRLP2
1420   E57B 20 8D EA            JSR CLRLIN      ;clear line
1421   E57E CA                  DEX
1422   E57F 10 FA               BPL CLRLP2
1423   E581             ;
1424   E581             ; "Home" cursor
1425   E581             ;
1426   E581             HOME
1427   E581 A0 00               LDY #$00
1428   E583 84 D3               STY CSRIDX      ;set column to 0
1429   E585 84 D6               STY CURROW      ;and row to 0, too
1430   E587             ;
1431   E587             ; Set screen pointers
1432   E587             ;
1433   E587             SCNPTR
1434   E587 A6 D6               LDX CURROW
1435   E589 A5 D3               LDA CSRIDX
1436   E58B
1437   E58B             SCNPLP
1438   E58B B4 D9               LDY SLLTBL,X
1439   E58D 30 08               BMI SCNPT1
1440   E58F
1441   E58F 18                  CLC
1442   E590 69 16               ADC #$16
1443   E592 85 D3               STA CSRIDX
1444   E594 CA                  DEX
1445   E595 10 F4               BPL SCNPLP
1446   E597
1447   E597             SCNPT1
1448   E597 B5 D9               LDA SLLTBL,X
1449   E599 29 03               AND #$03
1450   E59B 0D 88 02            ORA HIPAGE
1451   E59E 85 D2               STA LINPTR+1
1452   E5A0 BD FD ED            LDA LBSCAD,X
1453   E5A3 85 D1               STA LINPTR
1454   E5A5 A9 15               LDA #$15
1455   E5A7 E8                  INX
1456   E5A8
1457   E5A8             SCNLP1
1458   E5A8 B4 D9               LDY SLLTBL,X
1459   E5AA 30 06               BMI SCNEXIT
1460   E5AC
1461   E5AC 18                  CLC
1462   E5AD 69 16               ADC #$16
1463   E5AF E8                  INX
1464   E5B0 10 F6               BPL SCNLP1
1465   E5B2
1466   E5B2             SCNEXIT
1467   E5B2 85 D5               STA LINLEN
1468   E5B4 60                  RTS             ; return to init routine
```

CINT1 calls one routine, IODEF1, which resets the default input and output devices to the keyboard and screen, respectively, and then resets the VIC chip's registers to their default. The one interesting thing to note is the PANIC entry at $E5B5. The Kernal does not call the PANIC entry! The Kernal bypasses the extra JSR by calling the IODEF1 code directly. Not that there is any magic in the PANIC entry; it only calls the IODEF1 code and homes the cursor. I haven't yet examined the BASIC ROM, so it is possible that BASIC calls or vectors the PANIC entry.

```
1404   E5B5             ;================================================
1405   E5B5             ; PANIC - Set I/O defaults (unused entry point??)
1406   E5B5             ;
1407   E5B5             PANIC
1408   E5B5 20 BB E5            JSR IODEF1      ;reset devices and VIC regs
1409   E5B8 4C 81 E5            JMP HOME        ;home cursor
1410   E5BB                ;
1411   E5BB                ; Real PANIC entry; reset default devices
1412   E5BB                ;
1413   E5BB             IODEF1
1414   E5BB A9 03               LDA #$03
1415   E5BD 85 9A               STA OUTDEV      ;reset output to screen
```

```
1416   E5BF A9 00              LDA #$00
1417   E5C1 85 99              STA INDEV       ;reset input to kbrd
1418   E5C3                ;
1419   E5C3                ; Initialize 6561 VIC
1420   E5C3                ;
1421   E5C3 A2 10              LDX #$10        ;move 16 VIC registers
1422   E5C5
1423   E5C5
1424   E5C5                V6561I
1425   E5C5 BD E3 ED           LDA VICSUP-1,X  ;start at end of tbl
1426   E5C8 9D FF 8F           STA $8FFF,X     ;start at end of regs
1427   E5CB CA                 DEX             ;decrement index
1428   E5CC D0 F7              BNE V6561I      ;do next register
1429   E5CE
1430   E5CE 60                 RTS

2789   EDE4                ;
2790   EDE4                ;VIC chip video control constants
2791   EDE4                ;
2792   EDE4                VICSUP
2793   EDE4                        .db $05 ;bit7 interlace, 6-0 HCenter
.db $19 ;VCenter
.db $16 ;bit7=video address, 6-0 #rcols
.db $2E ;bit6-1=#rows, bit0=8x8 or 16x8 chars
.db $00 ;current TV raster beam line
.db $C0 ;bit0-3 start of char memory
        ;bit4-7 is rest of video address
;BITS 3,2,1,0 CM starting address
;              HEX   DEC
;0000   ROM  8000  32768 *default
;0001        8400  33792
;0010        8800  34816
;0011        8C00  35840
;1000   RAM  0000  0000
;1001        xxxx  }
;1010        xxxx  }unavail.
;1011        xxxx  }
;1100        1000  4096
;1101        1400  5120
;1110        1800  6144
;1111        1C00  7168
.db $00 ;Hpos of light pen
.db $00 ;Vpos of light pen
2794   EDEC                     .db $00 ;Digitized value of paddle X
.db $00 ;Digitized value of paddle Y
.db $00 ;Frequency for oscillator 1 (low)
.db $00 ;Frequency for oscillator 2 (medium)
.db $00 ;Frequency for oscillator 3 (high)
.db $00 ;Frequency of noise source
.db $00 ; bit0-3 sets volume of all sound
; bit4-7 are auxiliary color information
.db $1B ;Screen and border color register
; bits 4-7 select background color
; bits 0-2 select border color
; bit 3 selects inverted or normal mode
```

Conclusion

Next time, we'll examine more routines and answer any questions that you may have.

....
....
..
.

C=H #17

::

NTSC-PAL fixing, part 1
==========
Russel Reed <rreed@egypt.org>, Robin Harbron <macbeth@tbaytel.net>, S. Judd

Introduction
==========

Just about everyone knows that there are differences between NTSC and PAL machines. Most people are also familiar with at least a few of the technical issues, such as extra raster lines, and know that these differences can cause certain types of programs to fail, in particular games and demos. But how does one actually go about fixing one of these programs? It is that with which this series of articles is concerned.

"Fixing" has been a job which a number of 64 hackers have taken on, in order to enjoy programs written in another country. Some of the major game companies imported software from other countries and sometimes had to reprogram the software so it would run correctly. Fixing is not always a simple task, but there are some techniques and strategies which are useful in most cases. There are only a few different classes of problems overall.

We decided to approach this problem in a novel way: a pair of yay-hoos (Robin and Steve) would place themselves under the tutelage of an experienced fixer (Russ, a.k.a. Decomp/Style), fix up a program, and write up the results and experiences. The first step was deciding on which program (or programs) to fix. It had to first of all be fixable! Since time is always in short supply it couldn't be a big, complicated project. And finally, since we were just getting our fixing feet wet, the actual fixing job needed to be fairly straightforward (so the big custom track-loading demo will have to wait for a future article).

But demos are the natural fixing candidate, and after viewing several different ones, and examining the code to see what would need fixing, we finally decided on the demo "Slow Ideas", written by a couple of crazy Finns way back in 1989. It's a cool demo, was challenging and fun to fix, and turned out to be pedagogically a good choice. It's two pages, and each page had a number of effects in need of fixing. These pages, and the practical side of fixing, will be discussed in some detail below, but first we need to review the differences between PAL and NTSC, and discuss the implications of those differences, and how they manifest themselves in programs.

NTSC/PAL Differences

There is really just one primary difference between NTSC and PAL machines: the graphics, which means VIC. But since VIC generates the machine clock cycles, that means the computers run at different speeds. And since they run at different speeds, the CIA timers run at different speeds, the SIDs run at different speeds, and the CPUs run at different speeds. So just having a slightly different graphics format affects nearly every aspect of the machine's operation.

VIC and graphics

The PAL television standard is different from the NTSC standard. The primary difference is actually the way color is encoded, but the main issue for the 64 is the frame rate, the number of raster lines, and the number of cycles per raster line:

VIC chip	Frame Rate	Raster lines	Cycles per line
6567R56A (NTSC):	60Hz	262	64
6567R8+ (NTSC):	60Hz	263	65
6569 (PAL) :	50Hz	312	63

As the video raster beam sweeps across the television tube, VIC tells it what to display, one pixel at a time. The CPU clock is exactly 1/8th of the "pixel clock", so that one CPU cycle corresponds to eight pixels on the screen. Thus, it is clear from the above table that a PAL machine has 63*8 = 504 pixels per raster line. 320 of those pixels comprise the visible display, while the other 184 make up the left and right borders.

What is important here are the three numbers in the table, though. First consider the number of cycles per line. If a program is exactly synchronized with the raster, then it can make precise changes to the screen merely by letting a certain number of cycles elapse. A simple example is making raster bars; a more involved example is opening the side borders, or generating an FLI display (which you can read about in previous issues of C=Hacking). Needless to say, programs which require exact cycle timings will fail when run on a machine with a different number of cycles per line. (Note that on older computers, like the old Atari 2600, the CPU actually built the screen display, and so _all_ the screen code had to be exactly timed).

Next observe the different number of raster lines. The visible display begins on raster line 50, and there are 200 visible raster lines (320x200, you know). That leaves 13 raster lines for the NTSC border, but 62 raster lines for the PAL border. This causes two problems. Code which waits for a raster line greater than 263 will have problems on an NTSC machine. A busy loop such as

```
        LDA #$10        ;Wait for line 266
        CMP $D012
```

```
        BNE *-5
        LDA $D011
        BPL *-12
```

will never exit, while a raster IRQ will never occur. The code must be adjusted to use a different raster line or another method of timing. Sprite graphics on lines greater than 263 will wrap around on an NTSC screen and in some cases the image will be displayed twice. The sprites must either be moved or their images truncated to correct this.

Moreover, the extra raster lines mean extra cycles per PAL frame. Using the table above, an NTSC machine has 263*65 = 17095 cycles per frame, and a PAL machine has 312*63 = 19656 cycles per frame. In many demos and games those extra 2500 cycles are used to perform needed calculations and operations before the next frame begins, which leads to all kinds of trouble on an NTSC machine. Note that NTSC machines have more cycles available in the _visible_ display, while PAL machines have many more cycles available in the borders.

Finally, consider the frame rate. On an NTSC machine, there are 17095 cycles per frame, and 60 frames per second, giving 17095*60 = 1025700 cycles per second, or 1.02 MHz. On a PAL machine, there are 19656*50 = 982800 cycles per second, or 0.98 MHz. So although a PAL machine has more cycles per frame, the CPU runs slightly slower than on an NTSC machine. Thus a game like Elite, which involves raw computation, runs a little faster on an NTSC machine. But for most games and demos it is the cycles per frame which is important -- as long as all game calculations can get done before the next frame, the game can run at the full frame rate. Also note that most tunes are synced to the screen, so when a PAL tune, designed to play at 50 calls per second, is suddenly called 60 times each second, it will play noticeably faster.

By the way, there actually aren't _exactly_ 50 or 60 frames per second. The frames per second is actually determined by the machine clock rate, not the other way around! The actual system clock rates are 14318181 / 14 = 1022727Hz for NTSC and 17734472 / 18 = 985248Hz for PAL. Dividing by 17095 (PAL=19656) cycles per frame gives 59.826 frames/second NTSC and 50.124 fps PAL.

The important thing to remember here is that PAL machines run slightly slower than NTSC machines, but have many more cycles per video frame.

Just as a side node, the above calculation should indicate to you that although the AC electricity lines are 60Hz in the US and 50Hz in Europe, that has nothing to do with the 50/60Hz PAL/NTSC frame rates. Not only can an NTSC monitor easily display a 50Hz signal (let alone 59.826Hz), but the actual power frequency fluctuates around that 50/60Hz anyways, so that the AC line frequency is only 50/60Hz on _average_ -- good enough to run a clock, but not nearly precise enough to generate a video signal.

Also note that there are two different NTSC VIC chips in the table. The 64 cycles/line VIC was present in the earliest 64s shipped. This is actually a bug in the chip, though, and 65 cycles/line is the "correct", not to mention most common, NTSC VIC chip, and the one which we will refer to in this article. Oh? You don't believe me? Well, from the March 1985 IEEE Spectrum article:

> In addition to the difficuly with the ROM [sparklies], "I made a logic error," Charpentier recalled. The error, which was corrected sometime after Charpentier left Commodore, caused the early C-64s to generate the wrong number of clock cycles on each horizontal video line. "It was off by one," he said. "Instead of 65 cycles per line, I had 64."
>
> As a result, the 180-degree phase shift between the black-and-white and color information, which would have eliminated color transition problems, didn't occur. Depending on their color and the color of the background, the edges of some objects on the screen would appear slightly out of line...

There ya go!

Machine cycles

The tiny CPU speed difference has a number of important ramifications. It means that the CIA timers on a PAL machine run slightly slower than on an NTSC machine, so timer values may have to be recalibrated; note that the system CIA interrupt has a different setting for PAL and NTSC. Moreover, a disk drive runs at exactly 1MHz -- there are no "PAL disk drives" -- which means cycle-exact fastloaders will not synchronize correctly on different-speed machines. And finally, it means that SID works differently.

Not only will the tempo of any interrupt-based tune (raster or CIA)

change, but the actual pitch will change as well. SID generates its waveforms by simply updating an internal counter every cycle, so an NTSC SID is essentially playing a digital sample at 1.02 MHz. When that sample is played at 0.98 MHz, it's like slowing a record player down a little -- the pitch decreases. To be specific, the _absolute_ pitch changes, but the _relative_ pitch between notes does not; the tune plays the same, but the pitches are all a little over a quarter-step lower.

Practically speaking, this is totally irrelevant to fixing. Only the tune speed is of significant interest.

Fixing the Problems

The previous section described different classes of problems which can occur from the different cycles per line, lines per frame, cycles per frame, and cycles per second. These problems include screen syncs, bad interrupts and infinite busy-loops, too many cycles per frame, mis-timed timers and fastloaders, and different music tempos.

Fixing tunes is easy. For programs in which the interrupt frequency is otherwise unimportant, the interrupt timer source can be adjusted to the correct frequency. In other cases, if the interrupt frequency cannot be changed, the music speed may be adjusted by calling the play routine twice on every sixth interrupt or not calling it on the sixth interrupt. If perfection is needed, the music data might be adjusted or the music routine rewritten to work at the different frequency. Much of the time none of these are necessary, as the music sounds fine at the different speed anyway.

Next consider too many cycles per frame. In order to fix this class of problems, we must improve the efficiency of the code. There are three cases here. Sometimes busy waits are used to synchronize the code with a raster line on the screen. This wastes cycles and imposes some restrictions which result in additional wasted cycles. These busy waits may sometimes be replaced by raster interrupts which make better use of the available cycles.

At other times, programmers will set up all graphic updates so that the code executes in the vertical borders. NTSC machines have a big disadvantage here, as we've seen earlier. Usually this code can be rearranged to take advantage of the available cycles during the screen display. This can be tricky, as you don't want to be updating the screen at the same time it is being displayed, but by splitting updates up it can usually be pulled off.

Finally there are some cases in which neither of these techniques is of use. For a perfect fix, the code must be optimized, often by sacrificing memory. If it can't be optimized, then something has to go; effects can be truncated or updates slowed down so that less is updated each frame.

Next consider the different cycles per raster line. In most cases, this difference is not significant. The routines for which it is significant are raster routines, where synchronization with the video chip is established by using exactly the right number of processor cycles. FLI, VSP, and color bars are all affected by this. Color bars will have flicker and look crooked; VSP effects will often be shifted the wrong amount; FLI routines usually either repeat the top row of graphics all the way down the screen or else don't display at all. These problems can all be corrected by adding or subtracting the correct number of cycles per raster line. In most cases, this class of problem is actually the easiest to fix.

There are several approaches to putting the right number of cycles in for the fix. If the source is available, inserting a NOP instruction may be all that is required for an NTSC fix. Without the source, a modified routine may be inserted into some empty memory and the original routine bypassed. Sometimes the code may be shuffled around enough to insert a NOP with a machine language monitor. You can sometimes change the opcodes used to use a different number of cycles. Consider the delays that can be added with just two bytes:

```
CMP #$EA        ;2 cycles
BIT $EA         ;3 cycles
NOP NOP         ;4 cycles
INC $EA         ;5 cycles
CMP ($EA,X)     ;6 cycles
```

This gives lots of room to work with, assuming the flags aren't important. If .X or .Y is unused, then

```
STA $1234
```

can be changed to

STA $1234,X or STA $1234,Y

to gain an extra cycle (if .Y=0). If .Y is known to contain a fixed value like $FF, the target can be adjusted so that STA $1234 becomes STA $1135,Y. Usually in an FLI routine, you'll see STA $D018, STA $D011, and STA $D016 together. Two of these can be replaced with indexed opcodes to add the two extra cycles needed for an NTSC fix.

Also note that when sprites are active, a different amount of cycles may get stolen depending on which instruction is executing when the sprite data is read. C=Hacking #3 discusses this in detail, and explains how it may be used to synchronize code with the raster beam. From a fixing standpoint, it means that you don't always want to add two instruction cycles to convert a piece of PAL code to NTSC; sometimes, as in the demo below, only _one_ cycle must be added to fix certain routines, with the other cycle being eaten by VIC.

Sprites have a few differences between PAL and NTSC as well; like the other differences they are not evident in many programs. The horizontal sprite positions start at zero on the left side of the screen and increase as you move to the right. At some point past position 300, the positions wrap back around to the left side of the screen. For these high horizontal positions, the sprites are 8 pixels farther right on a PAL screen than they would be on an NTSC screen. When these high positions are used to create sprites that extend all the way to the left edge of the visible screen, problems show up. A PAL routine will have an eight pixel wide gap in the sprites on an NTSC machine while an NTSC routine will have eight pixels of overlap on a PAL machine. Often these positions are stored in a table or loaded into a register with immediate addressing. In this case it is trivial to adjust the value by eight in the appropriate direction. The vertical sprite registers present similar behavior, which is seen even less often.

As was stated earlier, the 65xx processors in the 64 and 1541 run at close to the same frequency, but the ratio of the 64's processor frequency to the 1541's processor frequency is not exactly 1.0, and the ratio is different on PAL and NTSC systems. This means that although the 6510 in the 64 and the 6502 in the 1541 may be synchronized at the start of a section of code, after executing the same number of cycles they will no longer be synchronized. Cycles may be added or subtracted on either processor to bring them back into sync, but this adjustment varies depending on either a PAL or NTSC system. This is most often seen in fastloaders, where the code depends on the two processors being in sync in order to transmit 8 bits one at a time or four pairs of two bits without the need for handshaking. As with raster routines, a few cycles need to be added or subtracted for a fix. Finding the right place to insert or remove these cycles can be challenging. Instead of trying to remove cycles from the 64 routine, which may be impossible, you can instead e.g. add cycles to the complementary drive routine.

Finally, understanding what problems may arise and what should be done to correct them is only part of the skillset needed by a good fixer. In the best case, you'll be working with source code which you've written and understand. In other cases, someone else may have written the code, in which case you must study and understand it before you can start trying to fix any NTSC/PAL problems. Often the source code isn't even available, leaving you to work in a machine language monitor. This imposes some new restraints. In a monitor, it can be difficult to shift blocks of code around. Several techniques are available to work around this. Code can easily be left out by replacing it with NOP instructions or inserting JMPs to branch around sections which aren't needed. Extra code can be patched in with a JMP instruction to the new code and another JMP instruction at its end to return to the old routines. Where cycles need to be added, opcodes may be changed; for example, a LDA $ABCD might be replaced with LDA $ABCD,Y to add an extra cycle. Finally, you may simply have to resort to a symbolic disassembly (i.e. disassemble into source code). This may become an absolute necessity in some cases.

Then there are the cases where the object code isn't even readily accessible. Some of the same skills used by crackers to remove copy protection are valuable. Most commonly, games, tools, and demos will be compressed to make the files shorter, adding a single BASIC SYS command line as well. These routines rarely try to be deceptive and are fairly short, so they can be undone, often just by modifying the existing decompression code a bit. It is common practice to have a sequence cruncher on top of an RLE (or equal-byte) packer or linker. Within the "scene", there will be intros to wade past, while commercial software will often have disk or tape copy protection and purposefully obtuse code.

Just remember that fixing is often about creativity and diligence. There is a common set of problems encountered and many creative solutions

to correct them. With practice, it becomes easier, just like anything else. You'll have an easier time of it if you pick your battles carefully. Remember that some problems just can't be fixed perfectly. If you get stuck, try a different challenge and perhaps come back to the problem later. And most of all, remember to have some fun!

With all this in mind, let's have a look at that demo!

Slow Ideas, page 1

Reconnaisance:

Page 1 is divided into basically three parts. The upper part of the screen contains a stretching sprite tech-tech extending into the borders, overlayed on top of raster bars. Then the rest of the screen features a large Pu-239 picture/logo. Finally, a bouncing sprite scroll takes place in the lower portion of the screen, extending into the lower (but not side) borders.

The screen is built using two interrupt routines, one located at $1240 and the other located at $1280. The $1240 routine is short and occurs at the bottom of the screen (raster $F8 or so). It removes the upper/lower borders, performs some calculations, and calls the music. The $1280 routine basically controlls the screen, and occurs at raster line $xx, at the top of the screen. It generates the tech-tech sideborder display, and performs most of the calculations. Both routines are vectored through $0314, not $FFFE.

Fortunately, there is a lot of distance between different routines, that is, there is a lot of empty memory between routines -- probably it was coded in an ML monitor. This means that adding patches, or shifting routines around, is much easier. What a forward-thinking guy that Pasi was, to realize that it would need to be fixed by C=Hacking one day.

When run, the demo is a mess. The most prominent defects are that the music plays very slowly, the screen flashes, with the main picture flickering between the top of the screen and the lower portion of the screen, the side borders are not open in the tech-tech, and sometimes the scrolling sprites are off the screen.

Too many cycles:

The half-speed music and flickering screen indicates that interrupts are getting skipped, which points a finger straight at too many cycles -- if the next interrupt is set *after* it is supposed to occur, then a whole frame will of course pass by before it actually occurs.

The first question is, where are the cycles getting eaten? That is, of the two interrupt routines, which is using too many cycles. The answer is immediately obvious: any interrupts that take place totally on the screen have *extra* cycles available -- 2 cycles per raster line. The loss of cycles comes in the lower border, which means the $1240 interrupt:

```
jsr tune
jsr blah1
jsr blah2
LDA $D012
CMP #$1E
BCC *-7
```

First note that curious $D012 code. It surely is meant to compare with line $011E, not line $1E. Line $011E doesn't ever occur on an NTSC machine, though, so it actually waits until line $1E, well past the $1280 interrupt. So the first order of business is to BIT that BCC out of existence.

Alas, this still does not fix up the flickering. The next step is to BIT ($2C) out the JSR tune call, to see if the problem really is cycles. And sure enough, ditching the tune gives a suddenly stable, or at least mostly stable, screen. BITting out the other two subroutines, but keeping the music, gives an unstable screen; the tune simply has to go somewhere else.

Finding extra cycles takes a bit of work; for now, it is enough to $2C-BIT out the JSR TUNE and focus on the other problems.

Tech-tech:

Let's have a look at the tech-tech routine:

```
$1280   LDA #$96
        STA $DD00
        LDY #$FF
        BIT $EA
```

```
                NOP
                NOP
                LDX #$09
                DEX
                BNE *-3
        $1290   LDX #$5F
        $1292   LDA $1000,X
                STA $D018
                DEC $D016       ;Open border
                STA $D021
                INC $D016
                LDA $1060,X
        $12A4   STA $D017
                STY $D017
                LDA $1100,X
                STA $D011
                DEX
        $12B1   LDA $1000,X
                STA $D018
                DEC $D016       ;Open border
                STA $D021
                INC $D016
                LDA $1060,X
                STA $D017
                STY $D017
        $12C9   BIT $EA
                NOP
                DEX
                BPL $1292
```

The routine has three parts: the initial delay at $1280, and a two-part loop, the first part at $1292 and the second at $12B1. The difference between the two is the LDA $1100,X STA $D011 at $12AA; without two parts to the loop, the branch would take too many cycles. Each part will need fixing, since each part uses at least one raster.

Opening the side borders requires exact timing, yet the above routine is entered through $0314, which will always have some cycle variance. The trick here is that all eight sprites are active; Pasi himself wrote a nice article in C=Hacking #3 on using sprites to synchronize the raster. The basic idea is to get the CPU to wait on a specific instruction; VIC then frees up the bus on a specific cycle, and you know exactly where you are on the raster line.

First to fix is the initial line delay. Since there are +2 extra NTSC cycles per raster line, it seems reasonable to first try adding +2 cycles to the delay:

```
        1287 BIT $EA    to      AND ($00),Y     ;+2 cycles
             NOP                NOP
             NOP                NOP
```

Really, CMP ($EA),Y is in general a better choice since it doesn't affect .A, but it doesn't matter here and we were young and naive, besides. If the above +2 cycles aren't enough, it will be clear soon enough (but it turns out they are enough).

Next up is the borders. Although the immediate impulse is to add +2 cycles per raster line -- +2 cycles to both loop parts -- it's not obvious where the raster sync is taking place, and +2 cycles might cause the sync to fail. In fact, what's needed is just +1 cycle per part:

```
        12A4 STA $D017  to      STA $CF18,Y     ;+1 cycle

        12C9 BIT $EA    to      INC $00EA       ;+1 cycle
             NOP
```

The $12A4 fix uses the fact that .Y=$FF. A NOP NOP NOP would have done the trick at $12C9 and be safer, but we used INC in our reckless youth.

And suddenly -- poof! The borders open up and the screen stops flickering. Why should the screen flicker at all? Remember that the loop is split in two because of an STA $D011 instruction. This STA pushes badlines off, so that the timing stays precise; since badlines never occur, the graphics data is never fetched. It's only after the rasterbars that VIC starts fetching graphics data; without the $D011 push, this data will appear on the first visible rasterline (hence the earlier flicker with too many cycles). When an imperfect (from incorrect cycle timing) push takes place, the picture can get the jitters. So now we know why the demo behaved as it did, earlier.

Sprite scroll:

And yet... the screen still flickers, when sprites bounce down

too low. Clearly the sprites are eating into the cycles needed by the lower border routines. The simplest way to fix this is of course to change the y-coordinates of all the sprites. One option is to re-do all the coordinate tables. But a much easier option exists: figure out which code stores the sprite coordinates, and subtract a fixed amount from each of the y-coordinates. This routine just happens to be located at $1590, and the simple insertion of code

```
        15AF STA $D001,Y to     CLC
             ...                SBC #$0E
                                STA $D001,Y
```

fixes things up just dandy.

Still too many cycles:

We still have the problem of what to do about the tune. Since there aren't enough cycles in the $1240 interrupt, they need to be found somewhere else, and the only somewhere else is the $1280 interrupt, during the logo display. The first thing to figure out is how many lines are needed, and how many lines are free. This is easy enough, by simply moving the JSR TUNE to the end of the $1280 routine, sandwiched between an INC $D020 and a DEC $D020. The border will then indicate the end of the $1280 interrupt as well as the size of the tune.

The good news is that $1280 has a fair amount of extra cycles available. The bad news is that the music is fairly inefficient, and needs even more cycles. But not many more -- just a good 8-12 raster lines. $1280 is pretty large, so maybe by rewriting some code enough cycles can be gained to make it all work. Note that if $1280 only had a few raster lines to spare, this task would be much more difficult (if not impossible).

Towards the end of the $1280 routine, there are a series of subroutine calls. One of them is a JSR $1E50. This code has two loops, one which copies values from a table at $1900 to a table at $1000, and one which ORAs a value into the $1000 table. Instead of copying and then ORAing, why not just combine the two loops?

```
1E50    LDY $1FC6
1E53    LDX #$5C
1E55    LDA $1900,Y
1E58    STA $1001,Y
1E5B    DEY
1E5C    DEX
1E5D    BPL $1E55               LDY $1FC6
1E5F    NOP
1E60    LDA $1FC1
1E63    STA $1E70
1E66    LDA $1FC2
1E69    STA $1E6F
1E6C    LDX #$5F                LDX #$5C
1E6E    LDA $32B6
1E71    ORA $1000,X             ORA $1900,Y
1E74    STA $1000,X             STA $1001,X
1E77    INC $1E6F
1E7A    BNE $1E8B
1E7C    INC $1E70
1E7F    LDA $1E70
1E82    CMP #$34
1E84    BNE $1E8B
1E86    LDA #$30
1E88    STA $1E70
1E8B    DEX                     DEY
1E8C    BPL $1E6E               DEX
1E8E    RTS                     BPL $1E6E
1E8F    BRK                     RTS
```

On the right are the patches we added, along with replacing the JSR $1E50 with JSR $1E5D. Instead of copying from $1900 to $1000 and then ORAing into $1000, it simply ORAs to $1900 and stores it in $1000 ($1001 actually, since that's where the first loop stored stuff). Sharp-eyed readers may have noticed that the patch affects $1001-$105D, whereas the second loop affected $1000-$105F; doesn't the patch above lose some bytes? Of course it does, what's your point? :). The ancient hacker technique applies here: try it, and if it works, don't touch it and don't ask questions! Better than a huge rewrite of self-modifying code.

Wrapping up:

With the above fix in place, and the tune moved to the $1280

interrupt, the demo finally seems to work great. All that remains is to save it and crunch. Figuring out which areas of memory are used is easy enough, by looking at the disassembler and the initialization code (which unpacks the code further). But after crunching -- uh oh, lockup. A program freeze shows that it is still running, but that the interrupts are not occuring. A glance at the setup code shows that $D012 is set, but $D011 is never set -- presumably the high bit is set, so the interrupt occurs on a nonexistant raster line. Adding a simple

```
        LDA #$1B
        STA $D011
```

to the initialization routine at $1443 fixes that up just fine, and the program decrunches successfully. Woo-hoo! One page down and one to go.

Slow Ideas, page 2

Reconaissance:

Page 2 has essentially four visible parts: a tech-tech at the top of the screen, followed by a swinging FALSTAFF sprite on top of rasters and open borders, followed by the ubiquitous Pu-239 pic/logo, followed finally by a sprite scroll on top of rasters with open side and bottom borders.

All this is done with two interrupts, one ($11FE) occuring at raster line $31, the other ($1350) occuring at line $D1. The $31 raster performs the tech-tech, the FALSTAFF rasters, and also performs some calculations for various sprite scroll effects. The $D1 raster handles the lower rasters and scrolling sprites, and also plays the music, scrolls the sprites, and does the calculations for the FALSTAFF sprite.

When run, the screen is quite a jumble -- too many rasters -- and needless to say, the screen effects need retiming. The music really sounds better at 50Hz, so it needs to be retimed as well.

Top to bottom:

Before fixing the timing problems, the extra cycles need to be addressed. In the $D1 interrupt, after the lower rasters are displayed there are a series of subroutine calls at $13B3:

```
        $13B3   JSR $211C       ;Tune
                JSR $0F80       ;Scroll sprites
                JSR $1300       ;Clear $7Fxx tables (used by JSR $0E00
                                ;  in $31 interrupt)
                JSR $1D00       ;FALSTAFF sprite
```

Deducing the function of each routine is easy enough -- just BIT it out and see what happens. BITting out the first three subroutines frees up enough cycles to make the screen stabilize. Finding cycles is usually more work than fixing up timing -- especially three subroutines worth! -- so it is enough to $2C-BIT out the three subroutine calls for now and fix up the timing first.

Tech-tech:

At the top of the screen is a normal tech-tech, controlled by the routine at $11FE. The code flows roughly as follows:

```
        $11FE   Set up $D020/$D021, waste a few cycles to get timing right
                LDX #$08
        $1214   LDA $xxxx,Y
                STA $D018
                LDA $xxxx,Y
                STA $D016
                INY
                DEX
                NOP NOP NOP
                Change VIC bank ($DD00)
                CPX #$00
                BEQ $123F
                NOP
                NOP
                NOP
                ...
                CPY #$2F
                BCC $1214
                JMP $125E
```

The thing to recognize here is that there are two loops -- on every eighth line the BEQ $123F branch is taken. A simple cycle count shows that the

first loop takes 63 cycles, and the BEQ branch adds an extra 20 cycles: obviously, these are simply timed for the normal and badlines. So all that is needed here is to add 2 cycles to each loop: I changed the CPX #$00 above to INX DEX for the first loop, and changed a NOP NOP to a CMP ($EA,X) in the $123F branch. With the raster timing correct, the next step is the initial timing at $11FE. By doing a little rearranging of code two bytes can be freed up, giving 2-6 cycles to fiddle with:

```
        $11FE   LDY #$0B        LDX #$0B
                LDA #$01        LDA #$01
                STA $D019       STA $D019
                LDX #$02        LDY #$02
        $1207   DEX             DEY
                BNE $1207       BNE $1207
                STY $D020       STX $D020
                STY $D021       STX $D021
                LDY #$00        NOP NOP
                LDX #$08        LDX #$08
        $1214   tech-tech loop
```

As you can see, by using .Y in the delay loop instead of .X, the LDY #$00 instruction becomes redundant. As it turns out, just +2 cycles are needed. Finally, there is the matter of the last raster line. When the JMP $125E is taken, there is some delay before changing the border and background registers, to get a nice solid line. It turns out that four extra cycles are needed here, and fortunately there just happens to be several padding bytes before $125E. By changing the JMP $125E to a JMP $125C, two extra NOPs are easily inserted. That takes care of the tech-tech.

Falstaff rasters:

Immediately after the tech-tech are the rasters and open borders behind the FALSTAFF sprite. JSR $0EA0 handles this part. We already fixed a sideborder routine in the first page, and this one is similar:

```
        $0EA0   LDX #$05                ;Initial delay
                DEX
                BNE *-3
                LDX #$15
                CMP #$EA
        $0EA9   BIT $EA                 ;Change to NOP NOP for +1 cycle
                LDA $xxxx,X
                DEC $D016
                STA $D021
                INC $D016
                STA $D020
                LDA $xxxx,X
                STA $D011
                NOP
                NOP
                NOP
                DEX
                BNE $0EA9
                LDX #$02                ;Change to LDX #$03 for last line
                delay loop
```

As before, +1 cycle needs to be added to the border loop, which is easy enough to do by changing the $0EA9 instruction. The end delay also needs a little change, to make the last line nice and solid (there are no sprites active on the last lines). Finally, the initial delay needs to be changed, to line up the routine correctly. I changed a CMP #$EA at $1277, just before the JSR $0EA0 call, to NOP NOP. As usual in this type of thing, it was easy to simply experiment with the initial timing until the borders opened up.

Bottom rasters and sprite scroll:

Finally, the bottom rasters need fixing up. Once again, the loop looks familiar:

```
          $135D LDY #$FF
                LDX #$55
                CMP #$EA
                NOP
                NOP
                NOP
          $1366 LDA $7F02,X
                STA $CF18,Y                     ;Stretch if necessary
                STY $D017
                LDA $1504,X
                STA $D011
```

```
              LDA $1C32,X
              DEC $D016                     ;Open border
              STA $D021
              INC $D016
              DEX
              BNE $1366
```

And once again, we need to add a cycle. As with page 1, .Y=$FF means that the STA $D011 can be changed to a $CF12,Y. Changing the CMP #$EA to EA EA adds the two cycles needed in the init to get everything aligned, and poof -- instant open borders (as long as all the sprites are being displayed correctly).

When the scroll is activated, something else becomes apparent: as the sprites scroll into the left border, they "pop" to the left by several extra pixels, i.e. the screen coordinates change by too much. To fix the sprite popping, all that was needed was to add 8 to $0F00 (the sprite coordinate).

Too many cycles:

Alas, the moment we've been dreading has arrived. Where in the world do we find enough cycles for three subroutines? One option is to get rid of some effects -- for example, the sprite scroller has many time-consuming effects that can occur; by getting rid of those, the tune can be placed in the $31 raster. Working through the code doesn't reveal subroutines that can easily be rewritten. Still, it sure would be nice to get all the effects going; but where to get the cycles?

A little thinking suggests something, though -- the PAL routine can't have THAT many extra cycles, simply because it doesn't get cycles until the raster routine is finished, at the bottom of the screen... oh duh. A glance at the raster routine shows that it covers $55 rasters. The interrupt takes place on line $D1, which means that the rasterbars extend all the way down to line $0127 = 294 or so, which is over 30 rasters past the last NTSC line. So suddenly there's a great big chunk of cycles, for free:

```
              (Orig PAL)      (Fixes)

        $135D LDY #$FF
              LDX #$55        LDX #$35        ;Only $35 raster lines
              CMP #$EA        NOP NOP         ;+2 cycles
              NOP
              NOP
              NOP
        $1366 LDA $7F02,X     LDA $7F22,X     ;Add $20 to compensate for .X
              STA $CF18,Y
              STY $D017
              LDA $1504,X     LDA $1524,X     ;Add $20 ...
              STA $D011       STA $CF12,X     ;+1 cycle to open borders
              LDA $1C32,X     LDA $1C52,X     ;Fix probably not needed
              DEC $D016                       ;Open border
              STA $D021
              INC $D016
              DEX
              BNE $1366
```

Note that the table offsets need to be increased; without that, the sprites start stretching all over the place and everything goes weird. With $35 raster lines instead of $55, the rasters go all the way down to the last NTSC lines, and free up just enough cycles for two of the $13B3 subroutines to be re-enabled:

```
        $13B3 JSR $211C       ;Tune
              JSR $0F80       ;Scroll sprites
              BIT $1300       ;Clears $7Fxx tables (sprite scroll)
              JSR $1D00       ;FALSTAFF swing
```

That pesky $1300 subroutine just pushes it over the edge, though:

```
        $1300 LDX #$2F
              LDA #$00
              STA $7F01,X
              STA $7F2D,X
              DEX
              BNE *-9
```

It doesn't take *that* many cycles, so one option is to use even less than $35 rasters. But this makes the bottom part look awfully small, and obscures parts of the scroll.

If you think about it for a moment, there are some free cycles

still hanging around elsewhere in the code: in the tech-tech routine!
There are lots of NOPs inside of the tech-tech loops; enough to piggy-back
the $1300 routine into it without much effort:

```
        $11FE   LDX #$0B        LDX #$0B
                LDA #$01        LSR $D019       ;Still 6 cycles, -2 bytes
                STA $D019       LDY #$02
                LDY #$02        DEY
        $1207   DEY             BNE $1205
                BNE $1207       STA $D020
                STX $D020       STX $D021
                STX $D021       LDX #$08
        $1210   NOP             NOP
                NOP             NOP
        $1212   LDX #$08        LDA $1887,Y
                LDA $1887,Y     STA $D018
                STA $D018       LDA $1B87,Y
                LDA $1B87,Y     STA $D016
                STA $D016       INY
                INY             ROL             ;Replaces LSR LSR ...
                DEX             ROL
                NOP             ROL
                NOP             AND #$03
                NOP             STA $DD00
                LSR             LDA #$00
                LSR             STA $7F01,Y
                LSR             STA $7F2A,Y
                LSR             DEX
                LSR             BEQ $123F       ;Every 8th raster
                LSR
                STA $DD00
                INX
                DEX
                BEQ $123F
```

Using LSR $D019 instead of LDA #$01 STA $D019 saves 2-bytes (6502
instructions like LSR use a "read-modify-write" cycle, and write #$FF
to the register while LSR is working, which is why LSR $D019 clears $D019!)
while still using 6 cycles. Replacing the LSR LSR ... stuff with the
ROL code saves another byte. Moving the DEX down saves two more bytes,
and getting rid of the NOPs saves three more, for a total of 8 bytes --
just enough for the LDA STA STA which replaces the JSR $1300 code.
The new code uses two fewer cycles, however -- but by rearranging the
initialization code we can just branch to a NOP at $1211 above.

Every eighth raster the branch is made to $123F. This routine
performs another INY, which has the effect of skipping every eighth entry
in the $7Fxx tables. Thus extra STAs need to be added:

```
          $123F BIT $EA         BIT $EA
                CMP ($EA,X)     STA $7F02,Y
                NOP             STA $7F2B,Y
                NOP
                ...
```

The two STA xxxx,Y each take 5 cycles, so 10 cycles exactly replaces
the 10 cycles used by CMP ($EA,X) NOP NOP. But since the STAs take two more
bytes, the rest of the routine needs to be moved forwards; luckily, there
are empty padding bytes immediately following the routine, so it moves
forwards without a hitch...

Well, maybe with one hitch. The whole tech-tech routine uses
self-modifying code all over the place. The modifying instructions
immediately follow the routine, and so need to be adjusted to the
new locations.

Finally, for some strange reason, the above fixes also affect
the FALSTAFF rasters. Adding an extra +2 cycles to the intial FALSTAFF
delay fixes that up, though.

So much for extra cycles!

The Tune:

Finally, the tune really does sound better at 50Hz, so a 5/6
delay needs to be added. There is some free memory at $09xx which is
ideal:

```
        $09E0   DEC $09FF
                BEQ *+3
                JMP $211C       ;Play tune
                LDA #$05        ;Skip tune every 6th frame
                STA $09FF
                RTS
```

As the comments say, it simply skips every sixth frame of the tune, to give it an effective play rate of 50Hz. It works!

Glitches:

Unfortunately, there are still a few glitches in the fix. The bottom rasters have some flicker in the upper right (and occasionally upper-left) corners. Pretty minor stuff, but still annoying. Fixing means rewriting the raster loop.

Much worse, though, is that the screen will glitch badly when certain sprite scroll effects are activated. This glitch seems to involve quite a lot of rasters, since even with the tune disabled glitches are still evident. One choice is to reduce the number of bottom rasters, but that doesn't look very good. Another is to eliminate effects, but that's no good. A third is to do a big code rewrite -- bleah. But the easiest is to just call it a 95% fix and say "good enough". Good enough.

And there it is -- Our First Fix. Hope u lik3 1t, d00dZ! Next time, we'll have a look at some of the routines which weren't covered this time, such as fastload and FLI routines. Until then, have fun fixin!

....
....
..
.

C=H #17

```
begin 644 slowideas-ntsc
M`0@+".\`GC(P-C$```!X+##0Y@&B4[WP")W_`<K0]Z*QO4`(G?8`RM#WH%;*
MO9I=G8CXBM#VSC$(SBX(B-#M3!4!@`*-``CF^M`"YOM@I/BB`R`O`H7XF*(%
M("\"(/D`H`"8H@,@+P+%^-#L(!$"A2U*T$`@)0)*D$@@)0)*D,[((!$"A2W)
M@)`+H@$@)@*%+2`1`J@@$0*JO3("X""0!HJB`R`F`J8MZ"#Y`,K0^HC0]_"L
M(!$"R?_P).D`H@`@+P*%+B`C`F7ZIBV%+:7[Y2Z%+NBQ+2#Y`,C*T/?PTZDW
MA0$L,-"E^H4MI?N%+EA,0Q1(K3&D[@("T`/N`P(JA?=H8.B*!O?0`R```I`2
MZ.`(T/+P"Z('Z`;WT`,@```(JRM#U&&```_R`#`00,$`(?,*#P"`]`@`8'L`HD
M8"[@&3]*<+;`3V_UA_)G\!>0DGS^DDZ/S]-W3I-]^)NJ8#A@0$H_UAG0"VQE
MA\`/P&)M)PGCPIN1\?/CXG?OA&4_/R+Y-$\^M-A/)/?WXG`"!@X>/IB>?>GO
MX)43(8#@"C]P&`/QF9FQTXQV!G;%7Q#YB/_G3C/&1\8ST[J$A$VD]^%-E-A-
MXT*"%9-Z462:(>$>B/!-$<R/)'<CJ3WZ_S]3D_D0">R:<<"GA@>'PJ:`,//W
MY5\)H!@*P7,-//7S]U64H__Y%2MSD5?Z.]9("=`8,&!"=?R,7X5>@2<ING@F
MX>$!P)J'J"0Y34Y!TYKT%!3T5)G]!@+PG0>,6%.G7@,!G(F\0>GQ,$!#IZ&R
M>])HMDA9EP@33HP<$]]U3DM2P"`P&`P",!(SX0=_P!<IWZ)"44G`GH$K#?0:
MIS7(F\W.`9D(J_T2-/04]/83$>`<@9\?`0,>```IBQ/CHB(A6/@Q,2$@<)TF
MRLWD3=5DW)N2R^$%7GYK,X%LK(WDZ?C\/`TZ0Z++H^/#`3P`0=9>'AP8:PX%
M9#)TI$):>"-9/0!6#N/OZ<C^?P'PGF)P@3@%50G,IP$9Z<[;J<!TC/R&<M,=
M.83I2^2U,14V*P-_#Y82A^^."RZ/@\6#)_#X.G@!@\.LNC/6F``K!+@.#P>$
MZ!Q\^>%ET+4]/Q`VGI`'ZE?TWC38_OY4?((DP?4,XZN1G2I2@0*$2R:'!7/"
MP`%>\5[ZAUK#KF+L?5Y7<K/C\MHU3\<'P>'J0E)4_^[`#7A'9V9F9V=CX]KQ
MLQV9]-RU95O(<#@Y7C;A:Q['_30KRN%TX5N=TP0&M,$`/K).W$JZ,5H0M(]H
MH'(*>JHA^N&TPK5PNR/<E*M+,08<??GA3*KT'*];KJC"<&%ZD*G%=S6\\.NX
MIAW<'<.ZJ!__EL2J+5%"E;!P='BB"<>%@T.+"D\'UD8:\=[$37OF`5YY4JJ]
M0!^:?])C^KS'\^H,9]/^/H[O]UE_^*]P)N\>]H2LR6T%1R\LDM/ZNZ#/@!_Z
M]>$`KWNQ&G!^/UE<V.4LJ\G__HIAP\>7&.ZS)(>6'JE9!T)+8V?EEO+9.40:
MWDKO6ZCB.6V$MC=A^?V=VY3+1HX61;MRA`]@`KLMMY2%KRLH*?VMBS4_<9%:
M[BZAI`UI=:LK&H!W5[[,@%'`_K+F+599+BW$+:V."EI=P,K+;F8<!<W,M>%B
MEN'131>*3-CHMLI;$P?H063$MK=1&_/"ZQ!UQ;AL'[,#MR:J==$.Y/]QJ2^&
M"O0/!*\LA`KRN[I,!!7D8"%QHX!JV_B,#!CQL>E<4];ELGS&_8#Y<:T1\!/Z
MXPP:'C+8D<!CP@>'<-B,&!BML^&*W-@.*X/V!^!ORL!JX#+G.%6\0'8L_"FU
M3.L;C'C8T9#UQQF9FQVL`S=ORJX,*ZM/E;SX&/8L*2E++S9F?,QZXIF<8-:V
M/`9\:>U<?D!\#'@"6:+PZN!FZP1>,13_W0!\28^0CH'>H00)X14_W<`CI$X1
M7U!M-?A7.%MT1`(!$0YK#&N>&,S;#Y`P4)$[]9:F*0M9`.9F8T77FL-C;S-F
M:Q%-C9CL##8XKJ$(!)R$_95&LDA+R`OZNB"Y?L!"@=72P0H+"@]2`0H%"@E2
M"0H-4F4*)$$(%U)1"B=`!]#\!/R$F)P(/#6(,8PL$"P0J!2DFJ">GJ"=XAGD
M5&A0;=`NSS/)]\LVPM#@$N#"EC^"\?WV&?87^8_`S'Y_X_`__'.\?[R'+_+]
M$_P4&Q0;]1KV"2>H!VCG-]<WYV8.++6V+BVM\_?<7W[/W\WG]OCO[(""@\'#
MPH"``8.`PT##@8`"PSQX!@6`!(*!@X&"A/?O"@F)"(@+BPK_R[XK0#<+0'<*
M1#0Z=FGBI@8T-W*D`C1K0J>J-%0.I,8T4`W'B,:C/-G<-QHBS["(T69]BK.Y
MGV`QHSH+%$/WO`(SH`"90A[IC/INFG@@212B`+T`5&=6!T:'OW6PCW7(C6L_
M<G`T.IG5"3K8ZZLQG5GLLN\@3A18:8&11I"@+K-T&<80ZXPP$[@D$9QU)NSI
M6$P-"&<`"Z;VG]TR_%$@4`E&B6A',:(QPP?[IH8P$&@00`F0``JVHH0*`\02
M`\S):8!%O@J1%PCLXG'**EA@*5K,/)&B3Y9X#R9`]3>!`SK=>K?H5,M&@&[0
M?YB`=751!.5H?M$OL51!R?V<+:$:0Z'<SZO6F`0F1C/C!=GN`.I7*?A<,:SD
M<</<R;RA##>1XQ-'Z($<6=3+N*.I86C#S*!B(P[T!-P?$"ZS]`!-AR)-G=A#
M^@']V"/UK/W)--`%J3"-9[]V,S]RF=(16E(FD@,:S[KF>;(H6C<!G5BN/<TC
M<<?BAAI)'*I5ZY"XO7)1>98_&YX_0`]R0#0P\E`T4O\RCWB!#QK3!'BVYP^C
MQC4>[`GW43_>P`VI!Q%\.3L`CZ@$?0`"H]!U'HCZ@$?0`#(+-^BH+T_&IW_U
MT[[)T_V;W?1YS^PL//?V4#?&NJMH@:M0I%)_47(`,SO!L3H1LSI@C]2>&?L`
MS]J&:0AT/N?ZQAG18!!H#U8PVHLAU\,.:+0<?#VV*DVA0&I<S75SMSVH"9>!
M`#3;RC""N$2!3%$P`$I;,,$PZ9B2B8^#\?Q`$`?W[]_7Q[>OGY>'?V]G5T\_
M-R<?#P;^[N;>UL:^MJZFEHZ&?G9N9EY>5DY&/CXV+BXF)AX>%# C[?8,CNAT
```

```
MPY8<.-#;R,G*AHYF;G9^AHZ6IJZVOL;6WN;N_P</'R<W/T]79V]_AY>?K[?'
MU]_O\`24?$16I-8HT2%9D'5H\:]T._0`^Z'?D@$CM(!`5,(%N3_Z`-)C]3"`
M52"E@2I7_5-!2/Q+N6/J.Y/J4@$$#1QVYX_,H^;`K&&T'&E'NFHJ,-,!,B`U
M6YB.@8&EB]^:LWO4N[6YR.LH4NOC*GZB!XH*J+U1>](78`:/L$"BX0T)D`T(
M3_O$:]S`"B.X.I/^9`="]1:3OP#WJ+2=)]#*$,Q0="^A&CFA'#N@P)_-)?J\
MP,3(S-#2UMK>XN;H[/#T]OK_YM\AH>)BHR.CY&2E)66F)F;G)V>H*&BHZ2EI
MZBIJJNKK*VNK^WDL;*RL[.TM+6U[ON[Z_6_KNYT<Y.<'-RQL;"OKZZMK*RKJ
MJFHIZ:EI*2CHJ"?GIV<FYJ9F)>6E).2D9".C8R+BHB'AH6$@H&`?WY\>WIY>
M'=U='-R<7!O;FUL:VII:&=F961C8F'MYU]>75U<6UI:65E85U=65E555.?ZG
MS_4N?_>OE%0^W_]/#N_*7C?\;/-15YGJW,I8Y_\PN7WFV#G_RS7EY?7^5+#S
M_8F)CY_LG/]EY_LW/_N_,_>OFAI^W_]/#QS>-_QL\U.9[%\&OPU<M%OX-?&W
MCAQT4$].34Q+2DE(1T9%1$)!0#\^/3LZ.3@W-30S,C$O+BTL*RDH)R8E)"(A
M(!\>'1P;&AGS>E\[R0\.#0T,"^WE">W;MX^WB[?`O;7=QWUX]W';7.3G!S<\
M\#>%!88&AP<'B`B)"@J+"XP,C8X.CQ`0D9(2DY05%9:7F!D9FIN<'1X?'Z"A
MHJ,D)28G*"BIJJNLK?B]]?O/[]'[\/S\7SX@/M]^/P#^7]]/WX/'IB[Q@"^7
MX!?;^^G[Z_/7UW[>('^KV^V%]K/Z?OTEOQ0WV=?DR?9:[Z:]''Y"4%_%]]P%
M\_@L,\?O/SN;#_'[S\[G<!]?O=S\O;X=?PX?6S\,_QC]T!=NW;R]L\_.?`%W
M0-S^!@[G]S]`PX#[N?'SZ\N?WCG_C:'FOW@Z%>/WGYW/*#^P,\?H.`^OWNY^
M7MZZ_7(%?#9^&?XQ_\1(>J'UF<P?E8<^?7E]+\9,DMD?9:OR*G"$RQ%7G>0[
MWT[WXN_>ZW\_+]?Y[F>_E_-,_#]A9W['SW,]_>N:9^'[RSLWA;&?A^^O_._>
M>>?A,?W%L9^'Y67\[\NYY^'=R;&?A^]\\3/7+<6QGY?OC>%L9^7[YWA'&?Q<
M@0'FH("Y^F@P/G>P2%9L9J%AF:AH<'YZH@(Y\Z)">SGAT;`UIG=(!X@'R$BT
MSM$Q.4%)`5%986D!<7F!B98V"`S-#4@-M6P%FYN&:-C%@'G!N<X^CH[(#P]/
MC\@0$%"0T31L(J,C[#82$G-4I+3$U.3U!.45)3(%15SS597H%A9T;"TMKFPV
M%U>7V#/586)C9&5F9VAI:FO.5LH&W-<;#<WN#CJV'(@<W1U=G9WSE>'EZ>WQ
M]?G\Z@('.6"@X35L0L-#Q&RV(F*<HMSC(V.CY"1DI/3:A+S!!,D$UKM,FY_7
M:7C_I:;5L81]%93U$=4U6SL_L;(@LR"TH+6VMZ"XSW<V+FZNR"\O;Z_SER0(
M[!PL/$4XUV1DIKE99!F9MLN+83\USL\@T-$@TEQ;&FL<7[94XOI!N[RG%[XN
M/DY>;GZ.GJZ^SM[N]?`6":^'CY>9`OA-CT]?;W7%L?"?Q<_-`#T0=<_`^,-N
M`/J<9:=`-"I&)T!T,K*J0V9)]"($.2B_U.-(.Z$<IQI!='("QI*KU[29'V+T
M4!0"219H#Y?<7T^ZJS&-Z5$:X#U;A$^F\>HE^M4"C`9,YLX"(=Q/(SI$F.<G
MZ==(DGO$&<Q1@)140?.L`A.N!/V`)^R!/V@)^V`@3FZZW%)>WY+T4K,,?J?^
M9)O97@&3!$I,&*CN)U)EH/1K"(&/U@`([;)N!%R`"!13X$2Q"9)\9!,7Q8"_
M4H@`8?L!!?L$G[A$_8`G[`D_8(GZ@$_4$GZA$_0`GZ`D_0(G[@(!-PG[A$_8
M`G[`D_8(GZ6\!-PGZA$_0`GZ`D_0)`Z"?_`4?Z%@22-[2*!.D_2?2"3Z0<K0
M\:U;'S.1K&D^JUA.'N3Z.#;4OACMJ7TQXT`>4P*@^0`#K]B-#TZNISLX40^R
MYBJ`'F?@_&&FD!B!#W9]!%56T3PP6?62R8XTS@=D'`F?B`FR`=I:<,R($#-@
M$^/9!=[WK2`KS(/\_H@>IH)T0&D.`U.A`UE"'DJ@A@P22&!XD*@`_@!3P^@?
M_HL__P/__X4?K\^?CX</G\C^315^^?_/*/UY_SQ1VGP`C?`?A1K*-!1E*,3X
M1<)AXL/U^6'[_JOT!1R*/U&]@*/T'Y^!1^A^'X*/U&,C#53GXH_4:C\H_87,
M1L(_<!'[P(_?!<G[P++KP`%3QKCPC]/$?HXN3]4>/PFG@FG!"KH0.=%)A_#_
MS^#^C#31#Q^(_?!8?2/11W/ZV'US(`H//\``>(\T$/P@;_$?7XCT_D=W\>)H\
MIIYK#Z4>B.Y5]/*L]/Y8;GXCN^%AE0"NQ<`17IB=B@(0YIV)BJ)`?Q5]`:KZ
M3I5T(B``'3L4;(`?IV4JTZEE@`Z\K``%P7J$T/@%7T#H_1]8>]`#\_\/\?\?
M\/\_\"C]?D-9\C]'D?H<C]!D?J-Y'$NS]?^?\4?JDUA_G\$8F`0(J#]>-^J%
MKQ0%(U4!Z%G^CU)+KRMD_/`.<X`<YP`YS@!SG`%E^J/11W*.@!SG``YSA9?J
MCT7D:U2@@CD28XCU1MCB-(XX%I^H]%7<CH5<0#CJ/UQP<?[XX[\]QR.+CA'2
MBT^T47O@`''`8;P`,'@`<#P`X"':`'@<`/`X`>$97@C$>$7QXBX'\!5]*/15
MW#BKB5;B-91H*LI&(&#*8=IC^F?R9V)F2F3ZP\?=,YDS<20`"YP%9,*Q>CG[
MOG/[?/?R?>_P__G@_^/A_X?`@1Q`:)\'^$2RD\T;::0*:<Z?RHDE(P`]&`/(
MP#Q&`<(P#Q$J//_`'C_@#A_X'L?\/</_.<#^A5/_!X'_@\!_X<`?\\1.(24>
M")!1%(^@.H6L0#\M8B5<1_^B_S_`?A_@_`_Q_@?X_P/P?X?@/\T5JM#,`!61
MF_+#*1B;#FCJ0H?"/K\.?]%"C]4?2/3^1W?RP[OA'=P'"8X)G`L/I1Z#B8N;
MN)`#\5IHL8%>`A__`Q1^O&?\/$?\?@/^?Q.^?[DC]\#Q_^4?OW.'__/(_=8?
M_^,/_\-&F`HQTZ`T3'@B2_'-5#?XXFBCZ<UB7RKNS_P<A_`X?$3!\B2'T?0?
M`'`/@#@&AN`?!$-\(@'!'T0/RG;?#X^.']_>/?W\>/CX3OM3#!,/U<-/`@,^
M%C&*&)@<H_5'^FY#Q_?WCW]_#CX^#`F]-2K$]&[_##_^#!_\#`_X*1Q&$30P
M!P`P`IT,+;]3%Y)+!2N$M?!?H>L/T>?T82F0937Z'J&50!ID:IB>6,3:L\A^
M!^%'ZLZU5G62LZO5=6*U7Q;U,GBU$"E3X1./E&#^H_7S\/WY'Z?/R)5'Z/GZ
M)*__`^?15[_P#YZ*OO\`-15^_``\>`H_0#!0=(O<`\`>!%=@(OX!_@0`_PD/
M7(A4R$)<@AA)$Q).5QX`)_``1_@`@_P!`?X"`/\"`'\#`#X(@>!&(D@$R<BI
M`6A4N-FUI^WZ/V;1^S*ZOVY1^_JU,!']P_'CY_'WY^%_#\`^'\`<'X"N28_4
M?JP&/Y77T`_"'11^^"&O@<$;J)G=1)S^^?Q\>?IO&"+,:GX\3OS\C[/#S_/!
MC_&!G_F%\;<`<&`P&/!X/(\C]X_QXQ_XQ?/Q@`:/E<HVI(2^AL`($X`<.`\^
M?/4?OCAQZE@%Y'Z@!#UC,X/HS#Y_@%J2"ANL$^P!@%7ZI4('@#@/6$DH^E'H
MK3!5RI]Q.G]XA`?$S^7L?KXMOTX5?J='"XNYF`.'`4``'#H3,='USX"/U5XI
M_)4`/GP"C&1F+B_5F3:C]7J=RK]4Q^^?`*@,&`++]3@?5V'_CRCT84?KPF(]
M%8GHY3Q'Z&++]3#Q'7\*.A<&4P+`=.`\`<!^`X#G!P#!%.<XZ+>.``\<6+W/
M"K]5]X+XHM^.<1^ABKZX4>G!X*TQ^`!6V:S8Q=[0K&8]7C$*/V933#33G5?3
M_A_RM:X'\'P#X!D0,`KI[^%A^O"Y:9D_D`I7C:AZ8/+!2.%5TJO?!>-^K(T?
M%AK(T%64C$`U&"!X``@P6&\X+9Z/%XWZLL!0K>"K01E*H4^3[`.#@`''3.Y5
M,`ZLZ%7ZJC$=R.A5Q)]E4GW'RP_1Q5^@*,99?JR\5IRI]42G\HN%23Q<9+?R
M?G*D9;?J?J1J?TRY9C6A;-[PNDJ587+W.E7G4JP`5RL<".3`#P'P#@/P'@)3
MELG"\[]1AO`<@(1=+CKT9%>%*.*GL#6>:J>+Y#]6.%NB`?U8L*GV!15`"P_6
M#P`^'@`<->;<;F`/@/"(#X6'[X!2,962H<CF0\<`<P(NL%8"BO3C6!`/IBX5>
M5JWP`RC9`4":L:$`_\`!*QQ0U;2`"P_5><FIET4?J'8'1^AR/T<5?HP/K&H]
M':F+;]5Z.P_@%@^L9VT+*&T`<$?K`M.5%(CKX!Q<<=KPL<VEX/%A^GP`N/5]
MRD`:!)7ADH_4%*O%=3%97K#`'X'P0">"C)7-^CRC#`5KA+*2`2IKI`63SIR4
M)'"F'8&,"S?_=3#'[B8!]<2]?+U?O];U42"Q>6[#Z2MD)[E^T]+I2`\#3H`+
M719`F\'[9F)IOWB@`RZ8%AZ@*?MK5.+P?MDZZ:ML6*;AQ673]#+9Z4`[H`8`
M#7Y5X84AR^YUHM^FP!ZL#KKF;(-8,7C@RNIC7BX;K5CX7&4E\N&&/W$!@>XZ
```

```
M!`#X*=?D7AQ`+Y'C7S(EZK&^%,0%\ANN2N!U@'>.,#_\+K%ZP\`'9LDQ[UQA
MN5W/&]9Q"$`X4_I!@JX%"P0.?V`8:!!S^ZX!#G]QW_C?WKF];W-CUI<S.9'K
M"Y^L$@H4'COL##`\<>AXV^+_\B_]<WK>YL>M+F9S(\8,!_WJ`\?\AS_[=.YY
M.Y[^[]H=WT]^3KZY_6_XV/&ES-\9'C"Y`=SSJ"AW/!W.?=O=X=^1ZY^[?&QX
MTN9OC([GM_[Y`>('Q">(?]$7ZE_$WR=\4'*/Q2^JG]%KXN?%YZO__>L#_OX`
M>,!P]7OJ[]?_>?!WGGWJ]\.'?R'X,[O5^5C\J7X37PD/A#%A<8&1H;>^WF0$
M)#14=(2DOWT'WGH/_/=IJWV_U7\/:7):4D(Z*AH2`?'AT<&QH9&!<6%).2D9
M"/CHV,BXJ*"0@'AH9Y[4D'!@4$`^W'OAGGJ!@@+YZ5T)"@P-#Q#SVKPN,C8Z
M/D)&2DY25_]\@(*FP.F!OX%$PP/%R`)9>@(F6$`@`LP"`Q;S:0##P0)#@<ZX
MB`!#`(%<=#BUI0<2`08)&#H@?-UTX:*B82!NNP*9W!X*'$`H"ACKHJF?`P*"
M;XM0L!F?`F*,P`9G=GMYM,*,2A-D()O:S%Q()%'$:6-J$`(L"D`R'BI`"L^!
M)BZ"O"J`P@M=?9`"!X<SC%V:K/83$'&<"B9^R&8^LP6&X>%LM0(/9FD)"VH"
M'`A`)`H:S^<4I"]IX/9!DP@*EID9G),0!H"P+9!L[&%XDS#=9P2\KEF6P:!6
M?+*$!%'7"S(`P%ETR5#R`#8!MN`RQB`A44>#(%ASLQF4H:0,]R`'9!LD&(9`
M\S>;>AP>!H:S4HWQE#7ECTK&"`*,`(&*!@RP[=>`N"O(19?.#T%!A1U=,T`T
M/+1</$@P$9;@+KI`7#.5.IPJ>5FSJ8N/$174L\&YER\NP4"6VV2XD,N2W<<C
M+UZ3;%!EH\$H3`1D_/VU&043(7E]T_CQMY9+=M0Q<!7%RQ;#B=R:(JS!!A5Q
M8.3N!78RLB`QG,\SL%1-XF$RC$W7B'O,W6-[-QCRP\!7$2\BX!#0D'`2UG``
M"C@-$WF8;TGF9[K*$&>;/@.`0QQU-MA[ET""#<M8`4CAX9IN@@99\AXRSO(&
M8@!2B;)$$8&,@Q1Y>%&&LF&4`FXD6>#S`X-9I51.]I(GHF\7'Y*AR+(0'B#S
M:UA\%L]WBPSLFBQ]QF`D`<"O,3BYQE/9B#!EUFAP6\P]=K4,O,#75X\,@4#W
MF'+HK=:H.ZN'74Q,8G.QX90/-H'0.$./,H>9+!H"Y?AP"N'AX?!:\4A=;SS-
M-X:C"49%V;#;T%+&P3<)H2\I4"=QT"(JYC]=?3E3`QC(R^<J\R:%01G@]7;=
M-`D"/*S>I\L^6!KE505U3A[IY?`&/!!!'4JVKS;9F&Z=QEF6WHWJX7"VK\>*
MH%&'K!XZD60("".-MTFO%>GO/\]S/$WE+MZL'X\3?"U3VU2]Y4[EI@=B@\8G
MC(V+C$R9^G)P<LJC$U+C`S;M@FU=F&SN>\UKVP'O-67.8<`AG"]+(7=$KAB<
MS9.V8`64KH[K79#,%@$29`O-/`%Q5@H6[$-AZR_!!42=A2ZGGG.6\M1YARZ+
MGRC@`N$]ZS>/>"!V%#+NE,3$K?"P;BX<4@&>FQHR.*`(9>Y8/%-V9Y0Z^&7+
M,4*#QEX:YB7F;I9`7AB%)2VM*Z3P=W+"^2X9,"H$&`K'P48I.&T9<6Q8^$\G
M689/DA7C#$\H038N!+F!!F\Z3Q,5;8!1-W`*SG$A5A72FCF'L89W.-7D>M&3
MFFH#9S`@J)ECV7'20[K;#@OB>5T\:GF2!49&IL0!82S:9-.:4W1\S<#01C&%
MN+)[7;P<F/X&6F1@ZN7N>SV*7E8"[LEV?H@S]9DAPH\WRI0P!"!1RF`>(#):
M[M!8]&Y<:/%XB"N?A*A`^4K7HMN9B7$B[LE8!"!:&O.1AU3#8V'`H6'L6@FJ
M=T^B8J&LY!)R-@H[N=%7>&RP[>=!X*&AC/@G92"#'4#!6YZ"EO%P^1RACE\*
M-Z&M4ZA:=[.]4X)7'NP\>;8O"#+'JVKP&0X/>8QLY'4<VV7/2)!@ERVK.<.%
M7):E!^4O&`UC32/0L94L^V<C*IT,/:R<7KMV7F"K.-W16\I!S-HSQ8G$%:]0
M%9`5CT,ZIX8%4\0T9\O2M?(QFCV9&'KF=MBH[MY>^!XP->:?N%('"'7J](84
M;=W)R&KVQGU.[K/-R!YX*8F7/6#5ZOB[K0>LNK@&#W,1+`-FLXR&=`:Y@_<,
MCJS><$[J?,2\R=LXXT@TN?#@\!+YZ07LJO&?L5=.[N3QU`AXD,N-1GJ[:]P2
MLEW$<*N9VP9BJZ`2QW5D[8$3>5N)B#D#G2&[Q774<[FL#'C%`ZA8/AP6U?CR
M;NV>=1;?"Y'#1.($2<W5LDW,)(HPG>:JA#O&.=TQO694/=V?.RU+'APE3P="
MIY/!8['@A;Q6\0`>OJI3:/_V?#?WQ.\KOW\`#Y:W`#UG`#V7\>[/CWE\>ZWC
MWQQ^@.`&PWL.Z?QW+FK<Y_#=$73YQH[K\#Y5O*ODF7.0);-85G-EJ>ZS+Y[*
M_@>LSX'_)8?^@9\FPW#E6Z<K^'K=GDS[T8&`?__A\OKYRC\YJ!ZYLEY9ZU7X
M93N'``];*0E&\]?+'%3N_-)?7,!O;.&?G/9]_*=U?!5;UP!UH'D^L&`WR'#/
MJ&LOL'R_S3N-*PVP``YI^FM/WVP`/9SZT#6&6#8!9#E%E#ZTOK]6!X5E1@6.
M!8YEH49R2RYH#*YDQK,M'`[]%-@%\D.7P8!V_3$KG+?+FL^L&R^P'+?`&H^H
M&D@`Y+Y0TUW;)``E`RT/1@%IX;%[?\!1\MZCZR[+[+<M\,ZSX"D]Q<GN_ZS_
MX;)(OA(X8!/K1)^\I/?6)]$T2^+G+___ZSX]3"M6N3)+'[,5U2`43?.\=]EE
M"'U4_\:(W\:)8_1._T=8%HZYWFQ^].WUT<7C&FE^P)%2`Z/EHJKV*M8U,KK
MV,Q])M=0`MS'6LS-6L0(>][#8[\O^^O8CGT46)`$UU$+\T9T`U.6GC[Y?]&=
M2/QHP8\XT7U#$N+:C!@%NA9_O`9_@,```0,P?]2!&IVM)^9(%X`^2`Z`(F"=
M<P=Q&[N(W=Q&XCAG%.=^(`U:R@C6?$-?WO`SX&+^HK6>F:SNT!2]9`1O2QOG
M6?757;S$XP!DSIL$P8L;QA[<\UMD_W@)9/_H"SAW9RO(J3.6N>$BF-,==&D]
M^4P'@'5KCY'YT4O_[C=<;KR0'EUG\Z;/BZPJHN:"UA;VYH6?=\XL02_O')O,
M7;&]A;@I\,GPT!^E9\,1A!P$(#,MG+D.N=,"[`@I[!P$N/IQF,YZ5UJ<&@2.
M-H$>FU.H[KW=>CCS&&GUCS.HE_$.L-QJ<&3@:`4X^&,7Q?.+UX837AH#GCX#
MZJ]/K:-7;&-:>:J)SW7N>[58TU_\MS`R8#0)O'P?G'1229[#6.<.T.-=I)\G
M><NPGDF*-?>]AL&B>IJ(,@H0=S+2=3+](G0&_N2BFUG4\G9]DX/Z^K!B_F0'
M4:)D`U+TA[H(UZ1S+GEJHX7O[U\MID%)UOI/H&)]/ULD7KT8_'NV([SO>3(*
M65+:S(%J3H/?M$Z&WZ$Z9`T>@5?J`8O4=Y8J"U)?(0E(PB2DY$>MZ;7_@"[R
M/L0`U9&5UHLSA1P^'#"W#!`EG]&D7_`EO_X^N7$'\0?1!U$'-1,&__C2`W("
MT0)Q`C02`G_HYDF.(07'(>2!TSQ\#O_O6$O@1/)283,82GP2F?\:$6+@+./U
MG+DB%@"WK%C0=?93_.%&]<+0AX![C*T`IRT3E>`-T4/3`:?[:%LKF`9]V=[Z
ML)/*^N8%GK:.-$#3%BRI6>X)X%&\1NU;4K:=9\4U_>,^\BY9Z##3ZZTWN5#S
M*RELO!+O01R7%XRFOMJVC@]F!9%(?&C+CG*6G!N^QR8(U?NMXDO'$QFVAQ\L
MPHT_9A%5K\LOXT^O\YFJ8!B<SX:;F=B9@6+B`O1UD,%;Q`S/&?YQ["G\SVO6
M7^ZSZL`F=9][S.UQRNK9Q='CI\C3UUP)51$&`[>B+!D$D#*M9T'':RR<<-:S
MD[64SC>3`BRO&9+-T6<<@PUV_;D;?ER%=EQ<\='@MG0'UT_-UJ;X)8,F(MFF
M$]G>H_)[V2,)91NG9>//@]Y7(8?>FF^W>R^L-QII!''S(;%YZWUZB7]4<.&7
M]<,0?9N&+]<+XU9T1Q0=-@,T0Q0--*3_N/&M9RN/AGWW,_=`#[F>$8',_1FS
MC@VSK-]S#F,,L8@0^XU*8KT0@SR+F;A<S>*K`%S6=\@"=K/LXRM#K>76%.NT
M(>UVX4&(3,0F8>.C:]8?M1(!+1"!I*/O"!\B3ACME;\LQIEPS&`*EL_#;*0>
M2\W2#PW6*V2),VX%)K=V[-HDAI>V>1([(FW7>'R4M/^Z(LB@-,E&>3]#Q"GA
MH"#(9E2S^*?!Q[7I??Z`U>@[D:&>CF+V]"K_I9$L^\D(L#!DSH%+.9R-3VYF
MFEX<P;(.9&?`-<=M:R`>,2V@!.%4O<KDY@_<9[F'-QIN8>7*H`/@=,Z8XU<<
MCE_:`)R?^0!J`$J8#0+8C9B*ZP#\`&T/:@`+EGJF)5HF*%I$.O^]*61K1R<.
MXF7V$QM`(J?^=)VF:,:4W'PL&T[,SU7OY:QL9:1KY:7OZ3WC2>[Z3W_B>\F3
MZ_R?4I:5H(0:AS&`;8@/L"N&HN>>'F(>(N17G24>EKRU)1Y&T-^9-)E_YGY^
M2%S;F03B%SO]R^)_N7+*,`-,O:Y@,%%F>Y&IP]WH:"&#A:?:/,]B-3AKO@,'
M"T^T>8[4>G"K'?I8>%H]X\PVH].&. 2P\/1[Q]@J]1Z<,=^EAX:BWS[!:SPY
```

```
M8[Y+#PQ%OWV#UGESQWR7'AB+?NMH&K//FC_@NSX9SUQSQC.Z!=G^>?`<(V/Y
MYZW#^>>1,5%A<9&AS9_@9DJ+"\R-3@[/T-'2T]465YD:G!W?H:.EI^HL[W)H
MU.'N_0(%;9CI2UM>;3:^K:WN+2WNNYY/EZ05=IW`%<GMSK7('=#ETHW3-=/E
MU(W2E=N5W*7=K<85Q7W>]\/Y6O%^\8[QSO'>\>[R-O)*\E[R<O*"\N;PNO>?
MV985][TZ7O[>A%\27QO?--5],WUY?7MALV`);"ML2VQE;!\"@H+?3P&'Y]_/
MNWR^]@MQOM\/"2?P"0$@X.$CK\2DY25EI>;S[@(``P=GX"./P%"''X$N/`&<
M?`.#!0%0`D`#(`.SX4`1`!C`(#0+`!*`04&]A1!`)EL@0069\;)@`(0`#((5
MG>"D)@($@@F^`*;/@(9X9G@`"3KHS(!#$V5R"$9GBAH3KH9L(5&SX$0C8$&P
M0>'P`N/AG1FD"%!@+<-JQP;<&P.@!06%XQ%`01F4"`,<`N/MU^!:$$!+R_7W
M?T@^[OC`V.#7ZWY(#0\-CPXX_G\=\1+/>+9^Q@['#KXO9^!=S(<'R@=+A[P^
M0^5>7L]/F&3R:&0R:!0WXGU_P`J+#`+&-WC(@SCY!I$"C`:.9\R8%'`SR^D-
MU_`3/R0=?+=X-O5+L_9![?46_&^HEQ]Q@S'/@^)!\F9)$A+&DD@2O#Y)8YY>
MST^6+!.AD[&LGAGX'TQ9!6+:&`@<<!@`6V\85(!0F)PCPEGE]2`^0OF\88CD
MBB&*2HI*W7Y_Y,$`83<WE.OIZ>>2S[>7@SO\<3!A&E`D@DWFA%'W".A(J^E7
MBN3K<4U@B"$<A^"3<EA@0@9A00QE0B/>""@=1;0@5!!BP\A887-Y!ZJ:A!A'
M4HEQ$31]J);#L"2,)D5Y-8<"2`3$\85P2&-1[%99$%).FZ(]U'D20Q85PA2C
MK($>D,8&4(9PR/8S-?`S\#!5\0BKW79W&_@<02K[4?AJO0^$:1DC0\"[P*_`
MJ\"GP)_`D\"+P'_`>\!WP&_`6\!C_P4$BX!1U5G^MO!>)U_^3$)]4(F02=+,
MIA99G0#A62N+S!59MA08LO,,6W,CZ1T!2D[,2P,(@R,Z@P0$E9<4P2/Q`2J<
M"M'L5NI7"%:/=1[(]U'^HZG,G)X(GA">**<9#0X$R@W1UX:D\$82YOR$,C!2
M4Z-I'Y*(\2HKAE)M&<0CV,5)94B/DV-T9Z2(='AXL/N$/C]'<>'T$CW1\H\8
M(ZE$<:B]59\,9%R(2@1Z(<EI0(:EM\02*$N4*:'=5^+"=A$=9F@\6&.7+"=1
M;&IN>(_5E]*)N"/CPZA$=1JJH$:<$L,!#`3$0_1$G_XX&%$)Y^&(06$1,66G
MHI;481^C8T+?ZEDPP/@A&$$P=258L%BZV\8)1H00F)"(?_Y,(X@&"292/Q2,
M2/15\JO-6)",.&"!2"`P5-VA!0*#(8."H@%13HNT>H=#!X9"!`>'!@1#(\5*
MGIN\(CT"O_#B"Q`2J"S)29Q(9J3.,H@N*80K+#`S.#<U,Z1!`<(%SI'Z>'""
M4\1O)(2":$4$T">*.I1^EBC])U'Z2*/T@4?HVH_1A1^BB-,R+E'Z4J/TF4?I
M$H_1U1^C2C]%E'Z(__CAX"$C&H#!#^H_%,>984INA70(?#<S1D_]!.([*!1T
M>/HK/48!X(*1\C"D]`=2Q[DK8IU5?;@J^W-0$J86\I6?:;EDH^A]'CE+#T48
M6'4<"3`5%<020A/$%<8P;)RF()X0F9^4G`8$A,\N4PA&")G&.P_B&`I@P&@C
M`ND"N$9Q,U7U<#DXC)QZF#D%"TNJ'AX&\(:NN40<8IRX]3)XOS_Y##1RE182
M,E67)88%D@SK+"YF24NW0I';HP)P:F9@7ES!%U^$Y(1D3K\(!T;&'7X+"@B'
MB(HY?L=(")U^$C'DL=?A<8,QW7X<'1X?PR"RY/T&$=?;K\/SPZ@F=)PQ_@C.
M4=4#H*AC=B2;FK/YS-J2,R"Z4+G7>>'Q^?'D,?';E;V-$(S7,V-\,[21G"\4
M-9WN,UGF\X_WBOT$W-O@]WK5T,##S@N9\Y+YO-FS!#Y.3C\(3,^6WF'R9GNP
MK</&0Y#UGZ74!9(%+*F@),.0&!J(&Q\:&WA\$`T+/;P'R`?8".7K>"+P&#($
MP<CLZ^')'L/V++%-X?!A&&$H\17-723.&?_!,42UB>6NM@W.3J&=P8,_F?3/
M5@H=,D'<#3PW2E^1NG)JZW34W_P(`!PYMQ`6W80%Q,#/S\'04$1*RL`<?H,!
MA(>'`(<S]"[<P;K```D)"&WB9L-74,'5PD`'^"`M/RL`P&O`%]@!U`(#P$*Y
MFN+7Y7/`%AX`.5\U+4`I6`8`/N>0/.N:30&V;RGDHU2(DZ:YW0`S;JGU0)O_
M8M9)S5#D5>!O'<R[JMH)>.B82#@H6(CY:AK+O*W?$&'SA54,I.TV?\F47RKV
MPQ!=J_I)&.BY!:ENM@%,F^L\D><^F?52LA5XWD>2Q')(O[;.!7STC$0\-$R$
M_78.QZ"YTPQE_YE3+2=!F_9Q"5\:AO-@3;NJ&0CXZ56I+O;`C%HK_=&6G?UD
MU$O+4N)Z&<%Q*.BP@%@X(!`.?\`\(+".\`GC(P-C'.'@L,,G#F`:)3O?`(G?
M\!RM#WHK&]0&>OL`./6@*,J]ER^=:>8^*T/;.,0C.+@B(T.U,%0&``8W@">;
MZT`+F^VF"D^*("("\"A?B8H@9Q\$'R`4`!,/#MQ?C0["`19S%J5H(!`2@25(
M)#P^L[(>'7DP$@%T0"0$SCU<>%0X_*J],@+@()`&BJ(#<=-,6]#BT8]_3(WW
M\*QQPY/_X$G2`40`ZDRX@(P)J%^H;L,2E^^4NAX]%B6R-/(D)OAIR_WKA(+*
M7ZE<<.583``02*TD:>Z-QD]P6!@15%2^[0P=$4#>]/(@``*0$NC@"-#R\*.H
M/T73M*J77J,,%)G_+@'J`P80#`X/@`(_B.`'#1CI\`4*"XG`H5?&=_A/@!JS
M!PAJ06-SV8"U3764XMWS^/07:`R4(#!`7=AF6("0H+V@8&[D#MX@>SL<]O+O
M$%-WV"NTY!87&!D:&QP='A\A(B,D)28G*"DJ*RPM+B_>H89LF)EN0S--H:FQ
MN<&Y`<G1V>'I\?H""A(:(BHR.D)*5M>Q+3$U.WB)Z@HJ2FJ*BJK*ZPLK2VN+
MJ\OL#"Q,;(RLS.T;Q&D#:MXC6V-OO]M7.7O]M9N_W_;6\`00/O]M=0N1<-#Q
M$3PNQ1!%UAD:01T?(2)!)2<I*RTO,3,U-SD[/$$_0T1X04>;])2TU/GU45)!
M55=96UU?86-E9VA!:VUO<7)!=7=Y>WV\2_Y>%.@S4!-,&?&81!AXD^LQ;`4,
M,;'GN9&3E9>9FD&=5GS_U4B3@_FIJ.JIH&J)J/H!JO^)JO>L?O['OOQ^KIJN
MT?J]'NO&B_O&`__1]-,>-'[(E(_1'_]1*CC^''K2ZQ%'[K2ZT1^[0<.V->]7
MU>U5U:3]H'[JSX>DO_ZJ^'NNBE=5\!=5]BCZW8],EDK<UM,".']E%+QT+NBZ
MKN`ZNNB[>NJ[/Q)(>C]#K#Z7=5WM/W4=U_0/U_4?[*+Q_?_WM,!I=94M'[#]
M*7H_8?EV<H.O''I^KL%#=U%_=U5?^M57^O55^`F;R_J+JOQ.*_C'7Z^\UW;3
M`UO]:U]]:@3:8*_&GU@GU4_G50&%]%A95=WH3.0U/\?KO'[OQ^[OI&WX#ZZZ
M+K;'HU;O_[*GMAGE.M'[#]K.LCB?4U^H#7VP$%V!!=%X!+(!L/T=5=0-CZ'H
M_5Z:_UI@2)1K2_J_*SAPRZJNKXI7]7`?"D+_V6JJKUE^_JO^KJNTMJ/+E;0.
MEK4>7K5>7G5==X_<JG7TP%K'5;Q(9>U3&Q0':U4U?QI:]&9&,NV7U[1:/@/K
M;HNM;JNM46]-5HKR[`JU^O^7ZP*`&N7`#,$_:;5RP`8P\M`',Y^U%R`"FNJH
MHA2I0!D!?YFM/W41D=65H>4YKN`0X8PYAQ<V1^6RSPZ0UBU(!&YWU(#&YW6/
MFC<[F#?'148$WU6"5Y6`)E@"<6&LHCJP3"P.XBUB3?+5`H>V[N^#*EWK>H@7
M-33[1"K7#>J>Q7.%M"-(=#N8^BR#0O1(5C1'08Z$W^.B`V-J.3FCDZ<ZL#JQ
M/W`5P:R$`A&IH(W/>I&^-NY%<U6<2PI@3-,`=S?V@A;?Z(`H?Y5!/#MTB727
M=*4KI9ND"Z1[I.NEPI%DJUXA';6\#E&Q^M"9U!\]^$@+@],XP](((/`.DU*:,
MG_T`JI@(7_J18_,C%*H,G^\`W)_:/S\H_/N/Q@EG+X'GXI[\?C/GY%>EK'95
ML&_(H@Z]``\XTVIFSJ?.5E"'E6A0?DC$@%9/&?A29\%+4@I88R:GAC1`/<-<
M84TG#EL>;6P`6`+34TX+7*`7I"'SI`'WJJ&+I@`Y7K3'TT1].,?3@!]'`#Z'
M'0F&`20"(>$X$LT4E'#V!,*`_>->&,P,#Q4O\:&Z5G*U/T@4B<I`8"!!["%2
M(QHJ!U/]&BB?9ES&B6@XE^R#0UH*V$PL$`#H,Z!*YV$[U5'(U0`]$M\(,0JD
M%&C7CK[P)X\<".-#2605(]"_3#*`!4ZI`!4^@M/H&Z-#QJ0=/C963]VD]?>I
M4@,G/4@$+\*8%_L"155.6@&PCY](A&D*`IY)YJ2$_<!AL[D@@1[@L1K4^LF`
MT.AMC`!6!.A&<@@EX3HPL<*V"=*&B=,'"=.-``'V1.I&A.J&Q.K'!.L'0'7X
M!AEKIW"X?A$9].D^?I5GZ6Y^F,]L\_36>W.?I[/;_/U"S]1<_4C)IS-;22E]
M.40!%)#*>$ ?A,#V@J3='2B@&HU+'*$/>B[VF?83V^G\PO-^B"U)PSH4`%$,
```

```
MQAD&@S&!1!#,Z^\.,1HQH)]#<:+:&05)_E('&@&ZCLS/IO,E2K_*\`V>=8!?
M*0U4Q<$B3JDL@2:]!6IZ_8RZIC-CDQV,JB!\A28(B3C!<D\&LPS=5$'5,NBU
M=#;%-4Y#&ZG*Y1Y!0!W<)B5:G[&C(EDK>PD[A02B[AI/!/W)'-`,J1:-3QJ1
MF-8\NIF9-K;:JFM)00&X'+&I(?0=T:F%:$:FAK:):HJ!:CB4V4](`!N:"B$&
M8JR9T`T&2PJ?].JI&$73]2EC1UA;5F"`GV^IU$5SN$LZ6O_&\P;+T@S2T\`]
MKZIP`YTG8S/%B))6T")M!29`]3'Z`,I"I4'(W4'&4`K5*";:>CG'TR*R?#AP
M)J#.F@XREEIC`#_T1JO&KTB4_S(QGQB0EZ2"LR)9]ZI6'&OADV36XXMG0`KF
M/7G4YQ(MQ#O'#QW,2B?K*W7!SC4]R#@"2"`#RP`3W`!VI!FF``88*D.0N<$S
MZ3Z(J3=@\..D!:"TY!=H1K4KU)A@)K\8!6[*;;!MX%6X+>);X(6^"0I<](8M
MXUN:W1;B(LY`0FF*)_BHJ+,Y!(P,I_C0T-C8XG^.CH\/#Z?ZT(4_R)/\C(R1
MGRJ?Y,G^N_$Z[2\4%*/.FO]>?BM>OBQ?P?BUTUW#+<,!)]^*6K!JH8H4G%9N
M"2#$6Q"K1A[1V1R8;4:$9$8$6Q=/\6101,1'-O-$!#4P/7.:&`8A4MZ$(@A!
MM<X`J""!B!`"L7SVF`F$EFEJ_BWPL`D3_@(*3^3]!TA%W+'RUY)_,<ZZ!@!;
M1@')_)^@8<!M<#'RUY)_,<ZQ][`7]\OC[5]?#PI_<37\\G_R.W]QI/V9\2C^
MZ$_N1\[^ZG;^XWS'+W?SYFQR/A?W0^I_<SM_/(J,'C\]H/-_,GH]W^V94#SE
MROM[5^#70Y#07;^YB7B_NAR_GQF/BYWRO[@?&1F/F?W2U^ATT'M'V=I9/D.C
MYB0^0..@`\IW/";3J93XO\OR4?F4PBXL,=9$W&01F0-R\&ND\/"0\*"`L(#`
MD-"0X*8]0H)"VZXTEWD&A@<&AX:$-<['\%AP8GFQ_,9:935NFJ$.&,Q_,;82
M@',?S7.U_A+7$'M?P6U_)C-?R6*[XPQP_@MC^"F)OX)</IKR=.%C_2+8EG+^
M28)\TUI_-?[5.PGHWL?S&($AC-LMCD"G+D8@@YP=&,9I0@@EAK9KF?GM)!'Y
MR6PP*8T0AH_86G+&8RTRF3\8WG>I-KP:S@R+BXI;U++>"/CT@DQL9&,>,4UW
MQ,?$QT1&Q$9'Q<?%1S'K'1\9>X<";\9%1<2Q]#1$4QSQ4;$QC7G$Q29Z[X&K
ME-=C^:R&$,9+MC^B&!%?\<[<CIQ,7KA_+UZG`;NQ_1@$8$"A&L?S2;&NG@4G
M0Z9[IH.F$X6SA6-4C4XU]L8S@J13IQ).7)'N!8QC.`X@M&L!R;L0X`$4R>R?
M\:R=2:L0R;S6,T]M6["Z4R747%N'`U<M=[\^F#PW/N-UB?[./ZMXF/T'`6OU
M=RY>S76UR,;K6L\]P+G9^XVF)_@;'ZGXN;%@6OU=RY>SAUN'"XV*/'9"_GD`
MOWE)D`.MY-[30!'"&A1!S.>P<=(Q8!2?80J(IVZ([6C*RAAN&AV1AS$>0?*8
MK!J47^<VL+'S79@/+9.&Y!'9#73QY(/5A4LD:`ZT(T#S]!<];0?/T)S]"^/O
M'L/T(,`^RE^$D2#"_'E@,<WSH!G-]4OUDD&@"-[8]``6/RK,JLM.OB@`1AQU
M\M0P52`&F&1&/8,.[_%MWZ"@X6'BXZ2EYN@I*>I0JJFGHYZ7CH1Z;F-734,Z
M-"\L*RTQ-CU%3EA1A:G)Y?\U8:%@D#GHW-#$O+2NK*ZRN,+.VNCY!Q,?*3$U
M.3DU,2LE'14-!P#\^8I_OS_`P<-#]QV9WO[V[N9/?0V5E9FIO=7Z(DY^KM]0
M>=L<2'-T]73S<&QGXDDOBLGY.(?G9OZW6=J;7)V>\E8;N'R'AH2!?WY]XLH"
M"AHEJ-D97J9FYB4CXJ#?'5N29V%=6>'I:76+'IS>7Z"X8D,E7IS:V)83T8]-
MC$M*RPN,SI"3%=B;GF%$CI:>H_`@I*"<EY..BXB%A.@.!S_X4GEU<6UH9&!<
M6EE:7%]E;'1^B92@J[;4!D49^IK[%&MIYN/+^M:RCF9&*A,U7U^@83!*5FZ"
MEJ:NMK*JGHIV6CXB!>G1O:VEG9VAK;G%U>7V`Y-F'X1-H+BF>LT_WOCOB(S$
M69G)Z>G9J5CH9\<65934$V+"0>YS3QTC*C20^2E=C;WJ$C93K=G)J6DHV),R
M5?7OPTQ'_)#L3*4"_/1G7N])G!)_AX['JAIJJLQZJ::AG):0BH7-Y]XE8F0F
M*&JM+W4+6F:6ML:^KH96#+XK:*6BX!V976;T(&-G;'!T>'OD8,_X+?X`!``.
M';X`$?']P@[!U41$0?9IQ\<&"#,*$#,%/M/C3\&$GW!!6$%8(?A!^"U\!"<@
M0QIX4Y_#'C!`^$#N'0%X07_@RQ3@N5<8\$N_&&*FFZY>[MZI^DG^`FBMC>&[
MD`D5,!G4?/I#`<RQ+8@;Y-5OMJ[`LW:=4[_<MX@C9_/V@(.=@4"G48@G4G/U
M,X^B:GG81YX4I@[X43::L(]JV"5;'QD@7@#Y(#H`B91X=T8_=&/VGV,G.RB`
M0!I=J%EU%GQ#)%ZAY\/I/45N>FDU.[0%+W3)8U5:0RO=:/IN(=Z*<P!DS((D
MSV(I/B2USF)8?^@')$-2#A7#JS&DAC;_.N`D`LI'UIUY)]?R-UW5O'<@<=Q_
M.Z5R!X.X*J+G9)9$50-<[04?:.%_>$)@.5BC0J&)I\,3^@$D%V11HOJ,?3%)
MND%*G5P9.!H!3'PE4J,)J*FFCVP(HFM`FH^#^G?HENI^TSK\.T.Z[1A]SO\N
MS/),U@8BZ)_9W<6M#Q:YR8T<G'D".$L[KF((5*G0@FN7Z[.O773FZS$TZTDU
M9@-,@.E2E4E3(!J7J0V@K*O<D-ST54<^@$][=$',7.N[SZ!L^KP0FA4N]N,^
MTR]N2:F2)U5S"UM<3\R!:U)T9):5,G9H5@*0^9`\\UO.O&/2.D4I+G5L1`5!
MQ@90<9,T28=$];W7(/`%WD^Q`#;.K7)&-_(:Y,?#AA58;!D1G&0^4'BKDD5)
M6?\;IF(F#8X_4YF^36O4C--!K[*?YB%O6(\0\`!VT+U&=6!<WV?:#>X+5SJP
M5/;\]4+C@`TQ,(Z7/<$UX$VI-ZAR*!:D#Y._$.\S[N70F&'WPU?4Z?,[J2]!
M*=.$;.I1H?ZZ`U>H4'':GHP0WJ%?2`,$V2#6!>A%H%#'./+:[=<RFEPY@J0%
M4'O`,8[:D&=(3\T`)@.QV)-DH6-!DV&-1DWV')!D.3IO54Q,<=(%6/U$5'8`
M^"Q'9*.UHAL2"7D#"I=5Z\$T;KA([UYUSZURD==2^0J)4X8HM2/QHQK4XQYX
M@H`G36BB2`/0`EBB)A`21&>E@':@`;0]Z;F*O!,)H[:M7^7[K!(!AQZ[YW/5
M^-%NH@0UH`:9,DE*@%(>%=`3VNLEC64DC54DO=5D>[-'NK1[]L>[9'UW4H(2
M2;1M(93`52.$GDA]@9U%RG@R1D1<F"=.`0,%S%V(()<"MC)M7MUSN3-.K&I
M6I\#"FD-8,6`?F;")B-@/`X<KL"L!2I4ONW4(G;44P*6+6ZUR@B<MZ$$S'3>
MN)YNX/N9C)\2_-DDY&UC3D95C*,$.U8(K"JP5&</SB1Q:\S7!*43P^_+_U?X
M$P<+3Y19GN2C4X>[]#!PH:?:/,]A$:G#74_`+S\UR:=J/3BG?FRL+1[RCS#:
MD^7S7R3J/3ACOTL/#46^?8*JL\.6.5I\EAX9D2@C0P+#2B<CW)"@L,#0WL/6
M.S_GRS/7:R>-SOP).>>;@#W#Q-H97D6"%/$Y"HLB+C)I09G@/V2DY46%YD:G
M!V?H:/*10P!-HPHH^D<"_\PYQR`G)2PF7B``)VPG!R5/)I2HY-*D(&4E]R4+
M)ALF,"9`)FPF#Y0*#/K\@4UO(']]_'O[_%P?E6Y-UN3]P)4`!_B>@#'Z:`%Q
M?H.@H,TG>,X`IF-)"7?(P0I[BT(,-?F*FMJ+?[-0;K4`NOX`!0=7X35Q-$;G
MF.!_`T$!M\MWS7`Q"#7%P)^%Q4<N9S1]&<#((!IK`0!3A`M>O"Q/AG5%V%PC
MBGK@Q\>$`QX;(`F^(`8BQ\"2?``*0IP$"P$"!5PV*##@@C$(B0K<ODI?GZ[7R
MGF5O51R%SE\F#7RSVB'^R1R/(*#`KA^,YL;A^(4,>@,GPG(G4G@Q^)X*0@9P
M^08$!/_`B`#%D'%HH@(_Q%/]LS/".*(Y2PSKSW$=T1PQ'*2([!"U1Q_Y"*JV
M5RK%$$%QZ[J]_>XS;!KGW*"5@`A*%`6,#(`'!]L#)H!A1=;`K':(@!L!%"=
MHHBZ/L=>A:,(D$I,-VP2`#ANN#@T3>'EP+!K&6V;0BO`!!U%NB?8>J_G:'SU
M?Z>K#_9>TXP/#\4_OZ8'F1N<;J0N3/R;895\3`.,#_C\//%S`H1)Z<$^(!!A
MAAF#S-*&&&"4!/6(>>)X1@F(T#/#CYC'M.^PZZ.^9.>N<0RS`\30#,B&_)O=
MX<%I5&'@?QL6$1X1'V$0>><\LZ\*+6#`>&)R@/\$H"/:<<\24>TXH\H]H=!
ME^J\I0T=EI(J3X(=[1Z@7B@G56@/@]EA?U'Y0/&$X/\"H0F9]@(6B/9?1_^4
M5I_5TX<<=S4O_,JLS@,(RK#+ML9_F[C]0Y.VH<P)NMD^'\\JWRTURQ@'M3&Y
M[3PX+3PM.C#JT*9N\&TH^%^GEPC]#^/GD>9\\K<CORT9:/#_LN6\,M$D_M!$
M\J,JB7\GEM"D&3<PR=(`XI*4R;,MDAF87=`V5HDN*)E*0G"0D5B9.IK^.$XH
```

```
M(`>KNK#RTAIYS,B'J_Z/A\J_YG@_@]AU,6+_6BE<+5?%C7("`8!GD57_CU$$
M\;0HW>"<Z\%'Q\'T?_],CY#[#H!Q^2X0G_/L=#52#L?&.N^9MU5/@XG^P8'W
MPMX6LA>.C&1?P;%NV8J=7P<.[OH+JN,T,$F#!LDE<PU8-#1F$P!CYD>V'C'$
M;,>4:6`'S=W^N+^/&CPE8L)[S/?KH)7^=^<(?G_PG5&=G3SSGO<]^$X.#FK_
M.+,Z*@E&V_/"C"\P.S/?_D@=S&(?''^X`:'W^!P["9#L"W^N>)H6\,`%/P&A
M@<5>6`R.0:GT+M`O0.__V?Q]SWUGGXXCG_9ABYF9B8=LSC5-C;S-N^9CMC9C
M1.)DD!JI`7GY_0;<%M*GX7*UXO#.BQU)MP,"!N0DE0V'JLZ<CQ8B(YM.'![K
MPAMRZ$R7+X9`?5\RF!L15TEM_O*_HD)9:^.<SG)P##*$XY<'ACN7P@BO)!`_
M:ZE<')R=01A!X!^71`>J`-+.^DIFU!SR8<>>4YW'X_YNGQ^6<@%3E6"_Q/0#
M!P\?I/\?M`'^LW-G0>&3>0_I+E`8<'#EA]J;&>7,/F)^P'W6-#`^;G3)CHPZ
M0&D[&_(#X<"08,3%@O@C%,'`@"F#P9+(`::V.(8342CC:S$^T`.-=^,^=BQ:
M_Y\`/)BB,C(>'1A#P;,S%O<8*H&-FJ__,P\7.K8^0#QB[&^0,;\8D>WJBX&?
MRIQ^`R49Z\!X/:U4?EG50P,$^2;E4KGL[Z1*S-L0`L968S[.5FF#N2`O9+[*
M6ZW#\QD>7`Z89Q]<0*`D,#-#0PO(8,$93A<-5AZN&JQL#E[J^K[[`'_X`=52
M!TJ5E2=(`4\@&T\A\\\G4\\\EYVK_:I.X6I.0!D.0`5`=,PZ_Z_2-.7(!JJI
MN`+XL/.W1555/2JM(;W29?O7_J])'<5^G5T&7[JB0-5TP%K@)O2'4]6`X8=(
MY=#U==2Y=&J*J50+U(H#5H6+D_?VYL"C6HT`&,"]:`N[]O4@)K^W(,*ME&1J
M^BUAL2I+!`5+;KN+6C`5;Z#`N;]U9E\(`'\`QU*IJ_35IO0`/V/Z](5LFFNL
M]+T1`#__0HL++]#6E0JB+S/T/Z[DUE50M,"8)?!TFET_0VE0U@I'POW\3]Y*
M4_4:W4[CIK`>!``_1CDH9R#7Z=*0&OU*)7J8_:1P_:EI1=.ARW'"T1"=U`1!
M3;D'3]VD:TV7SOT]ZBNSK?!$#\`#UB?ZJE`8=4\;F?5#QY1&AZ+;G28!F0'4
M@&'[(&'[2&@":D>/*!L:AZ*.L/T#L/VK`%9P)5?JQ`7=^WS*`7.BL=:!>VP"
M!>M`>ED=--![UT_TNG^H@*GQ.0TZY$.D\@//TYR`!?N@-#KB5ZJ_"5J<E6GC
MLU^?)5]N%!^3"!K5:57]50/O/_]6,#^5(#]K`Y:@Z0\3Y[,R>[C)`ZI`X_>H
M`4_C(`TWCRU7NHW*N(?9>Z5``U>%0&L!J;4AN$W//%I-I^N7HY=#EN.50F`*
MO2@#ZEZ^Z8'^\LBNJK5I``.XA%S)B1U>AU3Y>N?J!]]=>I6LX;@#]#]+*&-7
M]4NU=[R=;?:X@+WL"^+$"D?[4U8[;Z:;-)HESDK%[975'Z(C)+WS`GS`EA:<
ME=9H=+OU^G7A6$C6?H-2JH#2;&3Q<P$WT`6_P;7Y;_$QJYL_F(`>AZT`Y%#D
M.OV3I./T-4U3?R7A@:7]!S8QC]_IZ]9'KK2TJUW7SR=T]*O!K!JIVX;@/2Q,
MR5?H6)$Q^\J@<K1XG7+?'[8CH`LTI``YZOT_YK#BCK-669R3-2XS`_?]`KA8
MYD.1)S)U)U)TAJO)4@7><8J6+#J?$<:?7[A+4G4DG,G2',]Q``:K\=8`8`&!
MSE?W>!\-WLM[R[Z4K7I>5X/UJ(`NA)\4I^1P_=!?Q#GA`.]4CO4U^M)[28PE
M:EPI6M]VTS5X<H#P>+;3]>K;=OUW/1Y*56I^2P&&GK'[+2)'J]=]ZR$@"K^<
M@?CKH'ZJI2%XN+7JWO8%\2CYS8?H;>Q@!__=^(K/:^WB`_;RX`*NFW9*>BC]
MI8.W8"]#IN7!`IX*:XYB56NO>]#]_ZY!K4<;B@#VXFJU!ZSR-6B6]_(`NW!`
MN"#-8_8ZKQ);]SJ2_Z.==#'Z:O+K`)K)O.,,D@G4ZJ<G2FG[5,?L@U].W6?I
MUN6RQH(IUO.@`<]<?M,U^AT\U^B?2KJC]M(DU6HC#J\%`(JKB:H/52KB_ZYP
M[F<3^ZR]NLP#_0J&[,YY<JAYBV=/^?NNKP*Y;ZQ]OO;`Q`_$@#56S"#NOWXE
MCW(/[X+E3]K,`/98_"Z_N@/'%Y"K2`=3DHK^7H8A^GIS`-L,0UUF@'T4D?>Y
M7ZK`4LN,(9J^+$!Z.[&2#G7TY=>H-6``#5K%->/JL$\T[6\+92HOI``GVX"7
MWY^GE#_K#I)PZL'NCWL<6V'>[`_7O(!WC`:T7<[GU3<`1X@39`08X25.'U5&
MYGRR-_P/\`,?]_Q$CM>,!#@X?`.'OX3#'X.!P.`KR@.!5A/[0#!!@@0YCY*#
MF8!Q54I,`P&!5HW53X.,<%GR:2X='E*`LI#`,+#B`HQI@_:]-G&((X`;9O0)
M_KR!@SO5*@(-R5CSG@^!S`_X`WZOE#?T!_P'^K8CH>G>E/FKP__L_]L__O-%
M!YRP2X'#;B\'GARA^ZSNV\V_DO+OMUD2)$A04)2G63K(##1$CA(B1@?>/.N$
MA0%!:PG%X7%Q6$V28H68H&0K3`0>`@8*4)`0*"PJ;Z4ZQRB$DB4"(N+"P<%^
M/\`*?($=(T:RK(.`0\.1WXO&Y>6E<\*.<3_3A""$`V;:CG_+&?R$?PA`]"'H
M"_\'S<>,B8$O88"!-W/8_9RROJ/[N_GS\W?Z;C.3(W!QL>-FWXKD=\S=^?@<
MMMPX,'`,P!>"1.45/VY)`0K=/94T&=R9ZF];^$;K7`,!GNHVL"R8BP8"P;(1,
M6K)R<.#G7C1!0D2%&LDR0DQ6EI:JF)BQ,C`8#S,R4#P/3"X,*,2<L9H&98."
MJ\"ST>2T^(335['^\,W-:A5!0D+`K*DI24E4B4I<=M4$[]8&$_VYO';7HR(_
MG"=^$0><37@__)G/>H%7_#OO*\X0]XPQ[9&W;&'7'''I$%`@8(?^`@2E%35"
MZTI,U\`,%G_N=@&`4I.A7P?:=%8A&)6S<,`PD*Y/BGV;'63$PR#S"#_"2'^M
MP]C-%A@"E\29)DR98]<II`3)$2)4;,NF#`QU33>";B`UD!*P:!P0^K$0^$@Z
M-07%:O\E*0*P;0<%B/\#_/]5@/QH<8'''0I8A3`,D"5)B9(B)$B),-WY@[Z)
M??[YO'$(!`/7P_@8,'C9]HN)ZFG'_\+!\[/"'%?>$Z^,">2`@32EV0&L\>8X
MP0ZF,5#/$^48)FG96GRD^,$COM:V#`6G#)Y,0X($!"/`X%.OZF'R91B_C_P)
MDMHC1JDB1$B3B+AJM4,K`P8`X'HR?]_.`R)("'.?X`<F3D1*3(@3(L12^R<6
M&>'_)!I(JJ9ICA!C@CQ`@Q4A1@@IC$R/&##A.6`7_CSY4AGP@'L]BA/W:F?O
MG#Q#.`?,7J_!2&>)L+A3'##G@UNAR=Z9\NC!"Q?0`KTX(=5Q3.G,&C_@'_JN
MS$"?$"C_IQ;]V`/VMK<=G_+`?]P$.$&/=B/I#=+_!=)])%2KA@+__?V0@$)_
M8>%FG!@LH%TN3^!"#]#"#\3U^KVO(3#`\7(`D$!0$3(!`!$A1/4)6@$,K3Y_
ML4M;@0!Z?H@*EID9G)<0"NVJ;2:H:'D#'IP:8FWG2D`0`AR60:`IV)W!R<("
M&B`*))Z@R>@8UT%`*,`8*("C&6<PT5$PD#(!=+\#7"&L-K-:F0I-`/!6,8%#
M"EJB='UP71S:NT\4QP1JG,C%'5IT83,2P#0),U(0Y.#E$9'D-3T$DNTK1)-4
M4T#6,MJ*1M6XIV!P*A4,H5IJBB;0R'BK6'R'KF!<%6Y%CE+B8&D,MR:=/T&1
M,`5N0RR=0B*A*.J9<B+8*@F$@SACI8BR+Y.D")@K2PP;(2L.;2RTW)OB3'5>
MF8QVHZ`X"XG(4(4?J:J5S"N*)5*E+%`$H*@HHY1+'ZU=IVB3&:!!XC'@R`F(
M`+09!_Q.C["!-/<2-$RUR8F1?_`!A[)T?2F`%3\$G![>`Y#$K"=DM@].=;3
M/!00)&D2?P#2[0I_0:M=C!4G2@<8I3K]A'A"YPY`<23V09$VNAR5F+!BU2"@
M]$0';+E[M!R;T%Q@4>:U$K'3#<2'=L`$:E"&SC0X373*#4<\-S;\@_D5F.+\
M3]^Q<I<@20.I$(P,DZE*",G,`LFR"A`&6I]LNP4&)@:$`@$C""26XNRKI2VY
MP:,B:>#$.!"KBTX<@^`AH*XE8@P)L@*'I9"@:Q8KF$LV3<;%)!073@`6WE94
M$/7#F539(0%!@]9(`N>]\T-QD7`7D;6B=S%-U4W[6GY:!_P&Z!"#2$;K'"A&
MP5(0M-`2PZ,QTI=@K"0D^5V(B+'Z"LOVE:),G\DFFVUQM:CE;D[P/C:"(V@Z
M*`DN87B?8@0?TJE>'%:MVW\9/I@W%0)E_+4_F4#/VBX&0=%!5"UEGQ</('++
MH+0=DXQ-N+UH*8GF,0L<"Q-!J6;(%H#93Q*QHA;;@4;&YR2D[@,(*:6F6^<3
M88OP@/$&5Y$PE+)PQCTV-"UREVS%#@]A\%6L$-`R%K022+:Z1`,9+89<I8-$
M`E/)(,"`6Y\RFGD7;#X)2G`1=S`,):<0P$#B=L7PP(M9,=H5!1E#M'X!=*(0
M20&F+MHE<DII2=*Q@6;GD;^QQG.-M[37+:/6*!BC2Y8ZGAF>"D<Q4!$AED>0
M\29+<"$G3R*$LCDP*60U"Q(/`DF10)<#L59#E(9'\:9.%AH<AV93C9."!DV@
```

```
M\N^PU=@NX2K8F)+A;G"V"OAPH`@8HXQS,2XF+JDJ>S#$XRI-SB,0(<XE2D.V
M<2#P2DV4N&8V:EP=F?L81>@0Z!0X#!77!7"DN!1KE9;H"9:W'@W'6RTPRR:"
M5[%6XDPRJ`8PSLF<R/#8:IHX3]<>6DLO7>&DMKB0#0O=<0(;#\Y,S%.IQ@W2
M:3Z9BY*S,U."!.LI=<*8"8#V6*<Q!:%$;'%6$8@\72B#F1;=U$X<41O_@-F?
M%X9PX//%^QY":"`<V>`>MP=T";DS"R#\!8(N$,7P>6,D6PE4G<*.LR"<N`0P
MD#"RE,K#]BW@I@MV@H!#<TX>V^[`8X)H")A;'8V/[9W.L0Y:W)(MABC$&!4U
M`E-[]"DFQPH%W$H67^8J$R2$I<'-BP%Q)N]5J_0&:366K%/<-<X+!-QI1&67
MXBT-7CD`3$8I`(!`F4["PX/%Q-T-?L`#`$:&4%V@UU&/W&&)(G]"2IEP_<1;
M(`)7N16V$[<CJ>X.S8I6ZQ<`+<WCB$S-N9T["(&YUM,K^X.H,XRZBB;,?@.$
M/5*HJL-8HD\T:L$0.S:Z@BN!A;7HU<-@"VRSF;KP"(!Z0P8S#T6/!=L=!P4N
MGZ4.>&*3LNO4+I4,V/-5:+PRYSV(LN9$@K,C8HYX)KAX%):Q0"3,4S)TMMPZ
MPK`8DT\N+7H>IZK.J+')@S&@#I9H*,Q"`SG<`U&MHTG6Y5X,*KK<QEA?9C36
MP0LO"G`MZ4BGG)MQ>K8H/T36[O7<+&I5JJE%&$D3+4&4U6\<)SN.0Z3:G[FN
MSF'CG8FQF@S<,X()KF'&O9*XA;L=;>-$L)4[&-.)Q(EDXPEVEFR+$Q!"D)2U
MK+7@C5L'ZT<:PA=/UL'F\"T&7J?S+90XB3('^I\%"4J328[+-LMP)N9NB8J(
M"0"GTEQ0W9D'-UPLNH'@C%:4EC%&.(,PK%H$,I"M8(R$B!"0N9*N=&#GA=B3
M-*;%K&)S!SCC&H'H;%C-+Y8L$M4#>%\+@+CAS`@JS+<'M3B2+&PCNDNR:QDQ
M(,L^2ZX`S%&0MT6))G-L&QM-20"Q0@Y:S=$YN8TSPC,TXN!,V@)Z-,2'I,@X
MOR)A@",I[EC3>#DDVE^UQ!C!D@V!A;7`RLS>-$#%"<C.?RYT*!.6[7FDS#;R
M:68\TA&Y;TPVN)N)E*"H3$W"-1/"4('4N0<5&4L7,6@$"2R#&:*GV1F9&Y+Q
MXY)V17`1)Q9D2<Z4V$#]8%$(TC2%PJO1@M9AI:K77),`FZ\%5N?<@9SIV8KF
MFU*A;":::C>=ZF,'A("R&JV.,UXKK:A!HFXQ%)@I\PE&-)ULV:>F$8'.?[\6
M1\829`3\241299%BS9S2TU@A;QK8=\G<@;0C7_6:UF\ZE=C=7J=86WD+)].;
M/,I\6/#FS_N,[YI0:VK[:YK88S@0ALY6M]WC(;F6%1(.=YUWAT2&].4L:;A`
MER66\4QK;;)9SP6D_&JYUPUWA*9O+2$!0YI-9/EPMDMW0*`A5K.=\;-H5<<F
M%MK61*35;71RMFW9;@62%':[:T'?$O%0YHO35#G?&.<X?W`&<96!VI=NCQ"@
M.<6=S',`L5Z'#CA["3H3IU-3#N1J3>`&Q/N-=36T[Y$KCA2H\-Y_@(Y+XZ[D
MXM0;..HZT;B@RW@!-=3A5MG*XVP`C7$US-R<-7C-TB;;G/(NNQW>\%<=+YK\
MUZ$PR]1.)A#M%(M/470D#A[K0P%MSKL13C<M;8\MY0[E?([C+A&K8;<UWST&
M9I`D-9R+F;U46FYB&L6A[</5.H(;!);D^3AN.HQK878VIY=8X@"$MNF\VHPQ
MB8",N&RSD!,Q10R;H$W5^W0FYU9<Q`:UN;</@:ZX+GZ%(')S@J>#KE5*@S,"
MKJ\K":B$S)$'M4Y`A"Y/1QDI!0,5#@05;*EUT!F\&KURVG6D9YC9C==<EEM,
MW)[=<#KQW*YJ(`P4.9.%QH@9J,*FQ<*D:V\:AG:*0.O1J>'>C`6TJ93Y/EFK
MIP$;RXV\Q)FN1(QE^@1CI<X9X<G$`\X-S4U,P`49CP!@DG2Q=8,&HQD;Q;4L
M/L1<]+YQMN)K&24#`MU=%O@#\E:/#JFMN,ZR1NN+.\N40UQHM';-?0*Q2A((
?HT;F)R7&)D7./!%,L0$LN>9&1T:G*_!/]O_^"7-__\N4
`
end
```

::

The Herd Mentality as gleaned from the musings of Bil Herd

This is sort of a story that was really a bunch of posts I did on compuserve back in 1993. The language is coarse, the English malformed and if you're a certain ex-Commodore employee you might be offended by the story. In that case, the story is probably about someone else and not you; my fallback position would be that the whole story is a fabrication and meant for entertainment, not to mention that worrying about things like 5 year stories written by burnt-out wackos is bad for your cholesterol. As it turns out, this is about a sixth of the sillyness that went on, this is more about some of the events that centered on the 8563 chip which I was being asked about at the time. Oh yeah, the content is also dated with a slight slant towards the melodramatic.

12-Jan-93 19:28:05

Coming soon to a terminal near you... the gruesome story of the chip that almost ruined CES (and the C128 along with it)

EXPERIENCE the shame and horror of being a Chip Designer at Commodore during the Witch hunts!

SEE the expressions on Managers faces when they realize that their Bonuses are at stake!

HEAR the woeful lamenting of the programmers as they are beaten for no apparent reason!

SHARE the experience of being a Hardware Engineer... stalking the halls in search of programmers to beat (for no apparent reason).

LEARN how to say "THIS CHIP ONE SICK PUP" in Japanese.

Find out just how badly busted up the 80 column chip was and how many DIRTY fixes were needed to make that all crucial show in Vegas on January 6. (Christmas, what Christmas). .

<Said rather coyly in an attempt to elicit any positive responses> Unless of course no one is interested..... :) Bil

14-Jan-93 15:37:59

This is the first of many parts as this thing went round and round during our mad dash to make the CES show. I don't even remember what year it was. The 8563 was a holdover from the Z8000 based C900 (the "Z" machine as we called it). The people who worked on it were called the "Z" people, the place they hung out was called the "Z" lounge and well.... you get the idea.

The most interesting thing that came out of that group besides a disk controller that prompted you for what sector and cylinder you'd like to write to on every access, was one day they stole the furniture out of the lobby and made their own lounge disguising it as a VAX repair depot. We were so amused by this that we stopped teasing them for a week. (But I become distracted....)

Now the very very very early concept of the C128 was based on the D128, a 6509 based creature (boo... hiss). The engineers on the project had tacked a VIC chip onto the otherwise monochrome (6845 based) in an effort to add some color to an otherwise drab machine. No one dreamed that C64 compatibility was possible so no one thought along those lines. I was just coming off of finishing the PLUS 4 (before they added that AWFUL built in software to it) and even though I had done exactly what I was told to do I was not happy with the end result and had decided to make the next machine compatible with _something_ instead of yet another incompatible CBM machine. (I won't go into the "yes Virginia there is Compatibility" memo that I wrote that had the lawyers many years later still chuckling, suffice it to say I made some fairly brash statements regarding my opinion of product strategy) Consequently, I was allowed/forced to put my money where my mouth was and I took over the C128 project.

I looked at the existing schematics once and then started with a new design based on C64ness. The manager of the chip group approached me and said they had a color version of the 6845 if I was interested in using it it would definitely be done in time having been worked on already for a year and a half..... And so the story begins..... (to be continued)

16-Jan-93 19:06:28

Looking back I realize that the source of a lot of the problems with the 8563 is that it wasn't designed FOR the C128 and that the IC designers did not take part in the application of their chip the way the other designers did. The VIC and MMU designers took an active interest in how their chip was used and how the system design worked in relation to their chip. I overlooked ramifications of how the 8563 was spec'ed to work that came back to haunt me later. For example, it was explained to me how there was this block transfer feature for transferring characters for things like scrolling. Cool.... we need that. Later it would turn out when this feature finally did work correctly that it only was good for 256 characters at a time. 256 characters at a time. 256 characters at a time?? I never stopped to think to ask if the feature was semi-useless because it could only block move 3 and 1/3 lines at a time. Did I mention the block move was only good for 256 characters? Later a bug in this feature would almost prove a show stopper with a serious problem showing up in Vegas the night of setup before the CES show. But I get ahead of myself. It was also my understanding that this part had the same operating parameters as the 6845, a VERY common graphics adapter. Not scrutinizing the chip for timing differences the way I normally did any new chip was another mistake I made. The major timings indicated what speed class it was in and I didn't check them all. I blame myself as this really is the type of mistake an amateur makes. I wonder if I was in a hurry that day. :)

16-Jan-93 19:06:39

It turns out that a major change had been made to the way the Read/Write line was handled. When I asked about this, VERY late in the design cycle, like in Production when this problem turned up, I was told "remember,, this was designed to work in the Z8000 machine." ???!!!! ????!!!! Shoulda seen the look on my face! Even though the Z8000 machine was long dead and we had been TRYING for 6 months to use this damn thing in the C128 I'm being told NOW that you didn't design it to work the way we've been using it for 6 months? Shoulda asked.... it was my fault, shoulda asked "is this meant to work"..... :/

Don't get me wrong, the designer was VERY bright, he held patents for some of the "cells" in the Motorola 68000. It just that chip had to work in conjunction with other chips and that's where some of the problems lay. Our

story opens as Rev 0 of the chip.... (what's that..... doesn't work.... OK,) Our story opens as Rev 1 of the chip makes its debut and ......(pardon me a moment.....) Our story opens as Rev 2 of the chip makes it debut..... <to be continued>

19-Jan-93 20:50:41

Forgive the sporadic nature of these additions. Now where was I .... oh yeah.... It was sometime in September when we got 8563 Silicon (or so memory serves) good enough to stick in a system. I can't remember what all was wrong with the Chip but one concern we had was it occasionally (no spell checker tonight, bear with me) blew up.... big time.... turn over die and then smell bad..... But then all of the C128 prototypes did that on a semi regular basis as there wasn't really any custom silicon yet, just big circuit boards plugged in where custom chips would later go... but you can't wait for a system to be completed before starting software development. I don't think any of the Animals really gave it a thought until when the next rev of the chip came out and now with less other problems the blowing up 'seemed' more pronounced. Also the prototypes got more solid _almost_ every day. (I knew to go check on the programmer's prototype whenever I heard the sound of cold spray coming out of their office.... later it turned out they usually weren't spraying the boards just using their "Hardware Engineer" call. Sometimes all I had to do was touch the board in a mystical way and then back out slowly sometimes accompanied by ritual like chanting and humming. This became know as the "laying of hands". This worked every time except one, and that time it turned out I had stolen the power supply myself without telling them.... If anybody else got caught "messing with my guys" they'd get duct taped to a locker and then the box kicked out from under them leaving them stuck until they could peel themselves down, but that's another story.) ANYWAY, when this problem still existed on Rev 4 (I think it was) we got concerned. It was at this time that the single most scariest statement came out of the IC Design section in charge of the '63. This statement amounted to "you'll always have some chance statistically that any read or write cycle will fail due to (synchronicity)".

19-Jan-93 21:12:05

Synchronicity problems occur when two devices run off of two separate clocks, the VIC chip hence the rest of the system, runs off of a 14.318Mhz crystal and the 8563 runs off of a 16Mhz Oscillator. Now picture walking towards a revolving door with your arms full of packages and not looking up before launching yourself into the doorway. You may get through unscathed if your timing was accidentally just right, or you may fumble through losing some packages (synonymous to losing Data) in the process or if things REALLY foul up some of the packages may make it through and you're left stranded on the other side of the door (synonymous to a completely blown write cycle). What I didn't realize that he meant was that since there's always a chance for a bad cycle to slip through, he didn't take even the most rudimentary protection against bad synchronizing. It's MY FAULT I didn't ask, "what do you mean fully by that statement" because I'd of found out early that there was NO protection. As it turns out the 8563 instead of failing every 3 years or so (VERY livable by Commodore standards) it failed about 3 times a second. In other words if you tried to load the font all in one shot it would blow up every time! The IC designers refused to believe this up until mid December (CES in 2-3 weeks!) because "their unit in the lab didn't do it." Finally I said "show us" and they led the whole rabble (pitch forks, torches, ugly scene) down to the lab. It turns out they weren't EVEN TESTING THE CURRENT REV of the chip, (TWO revs old), they were testing it from Basic because it "blew up" every time they ran it at system speeds (No %^$#%$# sherlock. That's what we're trying to tell you) and even then it screwed up once and the designer reached for the reset switch saying that something does occasionally go wrong. Being one of the Animals with my reflexes highly tuned by Programmer Abusing I was able to snatch his arm in mid-air before he got to the reset switch, with blatant evidence there on the test screen.

19-Jan-93 21:12:15

One of the rabble was their boss and (I have been speaking about two designers interchangeably, but then they were interchangeable,) the word Finally came down "FIX IT". Hollow Victory as there was only two weeks till we packed for the show, and there were 4 or 5 other major problems (I'll say more later) with the chip and NO time to do another pass. It was obvious that if we were going to make CES something had to give. As Josey Wales said, "Thats when ya gotta get Mean.... I mean downright plumb crazy Loco Mean". And we knew we had to. The programmer thrashing's hit a all time high shortly after.

22-Jan-93 14:17:32

Memory flash, I just remembered when we found out there was no interrupt facility built in to the 8563. I remember how patient the designer was when he sat me down to explain to me that you don't need an interrupt from the 8563 indicating that an operation is complete because you can check the status ANY TIME merely by stopping what you're doing (over and over) and looking at the appropriate register, (even if this means banking in I/O) or better yet sit in a loop watching the register that indicates when an operation is done (what else could be going on in the system besides talking to the 8563 ???) Our running gag became not needing a ringer on the phone because you can pick it up ANY TIME and check to see if someone's on it, or better yet, sit at your desk all day picking the phone up. Even in the hottest discussions someone would suddenly stop, excuse himself, and pick up the nearest phone just to see if there was someone on it. This utterly failed to get the point across but provided hours of amusement. The owners at the local bar wondered what fixation the guys from Commodore had with the pay phone.

Any ways.... To back up to the other problems that plagued the 8563. Going into December a couple of things happened. The design had been changed to support a "back-bias generator". This thing is generally used to reduce power consumption and speed the chip up. Well, something was not quite right somewhere in the design because the chip got worse. The second thing that happened was that both designers took vacation. Nothing against that from my point of view here 8-9 years in the future, but right then we couldn't understand what these people were doing working on a critical project.

22-Jan-93 14:17:37

Or maybe I was just getting to used to eating Thanksgiving Dinner out of aluminum foil off of a Lab Bench. Christmas consisted of stopping at someone's house who lived in the area for a couple of hours on the way home from work. Anyways, the chips could no longer display a solid screen. The first couple of characters on each line were either missing or tearing, until the thing heated up, then they were just missing. Also, the yield of chips that even worked this good fell to where they only got 3 or 4 working chips the last run. A run is a Half-Lot at MOS and costs between $40,000 and $120,000 to run. Pretty expensive couple of chips.

The other problem takes a second to explain, but first a story..... Back when TED (the Plus four) had been mutilated decimated and defecated upon, management decided to kick the body one last time. "TED shall Talk" came the decree and the best minds in the industry were sought... We actually did have two of the most noted consumer speech people at the time, the guys who designed the "TI Speak an Spell" worked out of the Commodore Dallas office. They did a custom chip to interface a speech chip set to the processor. Operating open loop, in other words without feedback from any of the system design people (US) they defined the command registers. There was a register that you wrote to request a transfer. To REALLY request the transfer you wrote the same value a second time. We referred to this as the "do it, do it now" register or the "come on pretty please" request, or my favorite, "those #$%&@ Texans" register. ANYWAYS, the 8563 also had a problem where the 256 'bite' transfer didn't always take place properly, leaving a character behind. This ended up having the effect of characters scrolling upwards randomly.

22-Jan-93 14:17:45

So to recap, going into December we had a chip with .001% yield, the left columns didn't work, anytime there was one pixel by itself you couldn't see it, the semi useless block transfer didn't work right, the power supply had to be adjusted for each chip, and it blew up before you loaded all of the fonts unless you took 10 seconds to load the fonts in which case it blew up only sometimes. Finger pointing was in High swing, (the systems guys should have said they wanted WORKING silicon) with one department pitted against the other, which was sad because the other hardworking chip designers had performed small miracles in getting their stuff done on time. Managers started getting that look rabbits get in the headlights of onrushing Mack trucks, some started drinking, some reading poetry aloud and the worst were commonly seen doing both. Our favorite behavior was where they hid in their offices. It was rumored that the potted plant in the lobby was in line for one of the key middle management positions. Programmer beatings had hit a new high only to fall off to almost nothing overnight as even this no longer quelled the growing tension. A sprinkler head busted and rained all over computer equipment stored in the hallway. Engineering gathered as a whole and watched on as a $100,000 worth of equipment became waterlogged, their expressions much like the bystanders at a grisly accident who can't tear their attention away from the ensuing carnage. I can honestly say that it didn't seriously occur to me that we wouldn't be ready for CES, for if it

had, I might have succumbed to the temptation to go hide in my office (checking the telephone). There were just too many problems to stop and think what if. Next time (hopefully) I'll try and bring all the problems and answers together and explain why I stopped to tell that rather out of place TED story.

30-Jan-93 19:27:11

No single custom chip was working completely as we went into December with the possible exception of the 8510 CPU. The MMU had a problem where data was "bleeding through" from the upper 64K bank into the lower. This was in part due to a mixup in the different revision of "layers" that are used to make chips. This chip essentially had one of the older layers magically appear bring old problems with it. Unfortunately, this older layer had been used to fix newer problems so we didn't have a way to combine existing layers to fix ALL problems. Dave D'Orio (start telling ya some of the names of a few of the unsung types here) did a GREAT job of bringing most of the IC design efforts together. I was sitting with Dave in a bar, we were of course discussing work, when he suddenly figured out what the problem was. He had looked at the bad MMU chip under a microscope that day. Later that night, under the influence of a few Michelobs, his brain "developed" the picture his eyes had taken earlier and he realized that an earlier layer had gotten into the design.

30-Jan-93 19:49:06

This would not be the first time a problem would be addressed at this particular bar. (The Courtyard.... If you ever saw the David Letterman where the guy stops the fan with his tongue, he was a bartender there). The PLA had a problem where my group had made a typo in specifying the hundred some terms that comprised the different operating parameters. Well the designer in charge of the PLA took this rev as an opportunity to sneak a change into the chip without really going public with the fact he was making a change. When the change went through it caused one of the layers to shift towards one side and effectively shorted the input pins together. Ya should've seen the seen where the designer's boss was loudly proclaiming that Hardware must of screwed up because his engineer DIDN't make any changes (that would've been like admitting that something had been "broken"). You could tell by the way the designer's face was slowly turning red that he hadn't yet found a way of telling his boss that he had made a change. Talk about giving someone enough rope to hang themselves, we just kept paying it out yard by yard.

30-Jan-93 19:53:45

Anyways back to the 8563. The first problem was relatively easy to fix, providing you didn't give a hang about your own self respect. The 8563 designer mentioned that the block copy seemed to work better when you wrote the same command twice in a row. I made him explain this to me in public, mostly due to the mean streak I was starting to develop when it came to this particular subject. He calmly explained that you merely wrote to this register and then wrote to it again. I asked "you mean do it and do it now?" "Exactly", the designer exclaimed figuring he was on the home stretch to understanding (Intel, at last his eyes unfurled), "kinda like a 'come on pretty please register' I asked with my best innocent expression, "Well sort of" he replied doubt creeping in to his voice, "you wouldn't be from Texas would you", I asked my face the definition of sincerity, (said in the voice of the wanna-be HBO director on the HBO made for TV commercial) "why yes.... yes I am" he replied. Mind you a crowd had formed by this time, that poor guy never understood what was so funny about being from Texas or what a 'Damm Texan' register was.

30-Jan-93 19:53:50

This 'fix' actually did work some what, the only problem was that no one told the guy (Von Ertwine) who was developing CP/M at home (consultant). Von had wisely chosen not to try to follow all of the current Revs of the 8563, instead he latched onto a somewhat working Rev4 and kept if for software development. Later we would find out that Von, to make the 8563 work properly, was taking the little metal cup that came with his hot air popcorn popper (it was a buttercup to be exact) and would put an Ice cube in it and set it on the 8563. He got about 1/2 hour of operation per cube. On our side there was talk of rigging cans of cold spray with foot switches for the CES show, "sparkle??? I don't <pissshhh> see any sparkle <pissshhh>". Anyways, no-one told Von.... but don't worry, he would find out the day before CES during setup in 'Vegas.

23-Oct-93 16:57:43
Sb: C128, The Final Chapter

Thought I'd finish what I'd started back in January of this year. I had been talkin 'bout how busted up the 8563, now we get to the part about how it got fixed... well fixed good enough... well patched good enough to give every possible attempt at the appearance of maybe passably working...

One of the things that got worse instead of better was something called the back bias generator. Now as much as I admired the blind ambition (as opposed to unmitigated gall... no one ever said it was unmitigated gall and I am not saying that here and now) of slipping in a major change like that right before a CES show, it became obvious that it needed fixed. Now the back-bias generator connects to the substrate of the chip and if you've ever seen the ceramic versions of the 40 and 48 pin chips you would notice that the pin 1 indicator notch is gold colored. That is actually a contact to the substrate. I have never heard of anyone ever soldering to the pin 1 indicator notch but I had little to lose. At this point all I did have to lose was a HUGE jar of bad 8563's. (One night a sign in my handwriting "appeared" on this jar asking "Guess how many working 8563's there are in the jar and win a prize." Of course if the number you guessed was a positive real number you were wrong.) I soldered a wire between this tab and the closet ground pin. The left column reappeared though still a little broken up! The "EADY" prompt now proudly stated that the machine was "READY" and not really proclaiming it's desire to be known as the shortened version of Edward. To fix the remaining tearing we put 330 ohm pullups on the outputs and adjusted the power supply to 5.3 volts. This is the equivalent of letting Tim-the-Tool-Man-Taylor soup up your blender with a chainsaw motor but it worked. The side effect was that it would limit the useful life of the part to days instead of weeks as was the normal Commodore Quality Standard. I was afraid that this fix might be deemed worthy for production. (said with the kind of sardonic cynical smile that makes parole officers really hate their jobs)

Remember the synchronicity problem? Remember the revolving door analogy? We built a tower for the VIC chip that had something called a Phase Lock Loop on it which basically acted as a frequency doubler. This took the 8.18 Mhz Dot Clock (I think it was 8.18 Mhz.... been too long and too many other dot clock frequencies since then) and doubled it. We then ran a wire over to the 8563 and used this new frequency in place of its own 16 Mhz clock. Now this is equivalent to putting a revolving door at the other end of the room from the first door and synchronizing them so that they turn at the same rate. Now if you get through the first door and walk at the right speed every time towards the second door you will probably get through. This tower working amounted to a True Miracle and was accompanied by the sound of Hell Freezing over, the Rabbit getting the Trix, and several instances of Cats and Dogs sleeping together. This was the first time that making CES became a near possibility. We laughed, we cried, we got drunk. So much in hurry were we that the little 3" X 3" PCB was produced in 12 hours (a new record) and cost us about $1000 each.

A new problem cropped up with sparkle in multi-colored character mode when used for one of the C64 game modes. Getting all too used to this type of crises , I try a few things including adjusting the power supply to 4.75 volts. Total time-to-fix, 2 minutes 18 seconds, course now the 80 column display was tearing again. Machines are marked as to whether they can do 40 column mode, 80 column mode or both. We averaged 1-3 of these crises a day the last two weeks before CES. Several of us suffered withdrawal symptoms if the pressure laxed for even a few minutes. The contracted security guards accidentally started locking the door to one of the development labs during this time. A hole accidentally appeared in the wall allowing you to reach through and unlock it. They continued to lock it anyways even though the gaping hole stood silent witness to the ineffectiveness of trying to lock us out of our own lab during a critical design phase. We admired this singleness of purpose and considered changing professions.

We finished getting ready for CES about 2:00 in the morning of the day we were to leave at 6:00. On the way to catch the couple of hours sleep I hear the Live version of Solsbury Hill by Peter Gabriel, the theme song of the C128 Animals and take this as a good omen. Several hapless Programmers are spared the ritual sacrifice this night... little do they know they owe their lives to some unknown disc jockey.

Advertisements in the Las Vegas airport and again on a billboard enroute from the airport inform us that the C128 has craftily been designed to be expandable to 512K. Now it had been designed to be expandable originally and had been respecified by management so as to not be expandable in case next year's computer needed the expendability as the "New" reason to buy a Commodore computer. That's like not putting brakes on this years model of car so that next year you can tote the New model as reducing those annoying head-on crashes.

Upon arriving at the hotel we find that out hotel reservations have been canceled by someone who fits the description of an Atari employee. Three things occur in rapid succession. First I find the nearest person owning a credit card and briskly escort her to the desk were I rented a room for all available days, second, a phone call is placed to another nearby hotel canceling the room reservations for Jack Trameil and company, third, several of those C64's with built in monitors (C64DX's??? man it's been too long) are brought out and left laying around the hotel shift supervisors path accompanied by statements such as "My my, who left this nifty computer laying here... I'd bet they wouldn't miss it too much".

The next day we meet up with the guy who developed CPM (Von) for the C128. As I mentioned earlier, someone forgot to tell him about the silly little ramifications of an 8563 bug. His 'puter didn't do it as he had stopped upgrading 8563s on his development machine somewhere around Rev 4 and the problem appeared somewhere around Rev 6. As Von didn't carry all the machinery to do a CPM rebuild to fix the bug in software, it looked like CPM might not be showable. One third of the booth's design and advertising was based on showing CPM. In TRUE Animal fashion Von sat down with a disk editor and found every occurrence of bad writes to the 8563 and hand patched them. Bear in mind that CPM is stored with the bytes backwards in sectors that are stored themselves in reverse order. Also bear in mind that he could neither increase or decrease the number of instructions, he could only exchange them for different ones. Did I mention hand calculating the new checksums for the sectors? All this with a Disk Editor. I was impressed.

Everything else went pretty smooth, every supply was adjusted at the last moment for best performance for that particular demo. One application has reverse green (black on green) and the 330 ohm pullups won't allow the monitor to turn off fast enough for the black characters. I had had alternate pullup packs made up back in West Chester and put them in to service. On the average,2 almost working 8563's would appear each day, hand carried by people coming to Vegas. Another crisis, no problem, this was getting too easy. If a machine started to sparkle during the demo, I would pull out my ever present tweak tool and give a little demonstration as to the adjustability of the New Commodore power supplies. People were amazed by Commodore supplies that worked, much less had a voltage adjustment and an externally accessible fuse. I explained (and meant it) that real bad power supplies with inaccessible fuses were a thing of Commodore's past and that the New design philosophy meant increased quality and common sense.

I'm told they removed the fuse access from production units the month after I left Commodore.

The names of the people who worked on the PCB layout can be found on the bottom of the PCB.

"RIP: HERD, FISH, RUBINO"

The syntax refers to an inside joke where we supposedly gave our lives in an effort to get the FCC production board done in time, after being informed just the week before by a middle manager that all the work on the C128 must stop as this project has gone on far too long. After the head of Engineering got back from his business trip and inquired as to why the C128 had been put on hold, the middle manger nimbly spoke expounding the virtues of getting right on the job immediately and someone else, _his_ boss perhaps, had made such an ill suited decision. The bottom line was we lived in the PCB layout area for the next several days. I slept there on an airmatress or was otherwise available 24 hours a day to answer any layout questions. The computer room was so cold that the Egg Mcmuffins we bought the first day were still good 3 days later.

About the Z80:

What court ordered Commodore to install the Z80?

It wasn't mandated by court order, it was mandated by a 23 year old engineer that realized that marketing had gone and said that we were 100% compatible. This turned out to be a hard nut to crack as no-one knew what C64 compatibility meant. Companies who designed cartridges for the C64 used glitches to clock their circuitry not realizing that the glitches were not to be depended on, etc.

The Z80/CPM cartridge didn't work on all C64's, and no-one had really taken the time to figure out why. Someone noticed that a certain brand of the address buffer used in the CPM cart worked better than others so someone concluded that it must be the timing parameters that made a difference. This wasn't true, it was a very subtle problem that dealt with the way the 6502, the Z80 and the DRAM had been interlaced together. So here we had a CPM cart that didn't work with all C64's and it worked even

less reliably with the C128 even though the timing parameters in the C128 were far better. In my opinion you couldn't call the C128 compatible with the CPM cart as it only ran 20% of the time when tested overnight.

ALSO, I worked hard to make sure the C128 had a reliable power supply. I was told "no fuse'..... oops one got in there by accident... in fact it was easily accessible... darn it anyway. However, with the wide variations in minimum and maximum power supply requirements we couldn't handle the CPM cart, it needed an additional .5 amp because of some wasteful power techniques that were used in it. I couldn't foot the bill for an additional .5 amp that might only occasionally be used.

SO, with that said, I accidentally designed the Z80 into the next rev of the board. We designed the C128 in 6 months from start to finish INCLUDING custom silicon, these were records back then, the Z80 was added around the second month.

The Z80 normally calls for a DRAM cycle whenever it needs one... it might go 3 clocks and then 4 and then 6 and then 5 and then 7 between dram accesses. Since the processor shares the bus with the VIC chip there are only certain time when the bus is available for a DRAM cycles. Since the shortest cycle for a DRAM access for the Z80 is 3 clock cycles, you are sure to catch a DRAM access if you do 2 cycles (wait for vic) then 2 cycles (wait for vic). Whether you catch the Z80 between clock 1 and 2 or between 2 and 3 doesn't matter due to the special circuitry in the design. Otherwise if you just let the Z80 rip it crashes when it tries to grab DRAM while there is a video(vic) cycle going on. And that's why it runs at a clock-stretched 2MHz. The REAL bitch was the Ready circuitry when flipping between DMA/6502/Z80.

The C128 design team: SYS32800,123,45,6

Bil Herd	Original design and Hardware team leader.
Dave Haynie	Integration, timing analysis, and all those dirty jobs involving computer analysis which was something totally new for CBM.
Frank Palaia	One of three people in the world who honestly knows how to make a Z80 and a 6502 live peacefully with each other in a synchronous, dual video controller, time sliced, DRAM based system.
Fred Bowen	Kernal and all system like things. Dangerous when cornered. Has been known to brandish common sense when trapped.
Terry Ryan	Brought structure to Basic and got in trouble for it. Threatened with the loss of his job if he ever did anything that made as much sense again. Has been know to use cynicism in ways that violate most Nuclear Ban Treaties.
Von Ertwine	CPM. Sacrificed his family's popcorn maker in the search of a better machine.
Dave DiOrio	VIC chip mods and IC team leader. Ruined the theory that most chip designers were from Pluto.
Victor	MMU integration. Caused much dissention by being one of the nicest guys you'd ever meet.
Greg Berlin	1571 Disk Drive design. Originator of Berlin-Speak. I think of Greg every night. He separated my shoulder in a friendly brawl in a bar parking lot and I still cant sleep on that side.
Dave Siracusa	1571 Software. Aka "The Butcher"

Not to mention the 8563 designers who made this story possible.

.....
....
..
.

C=H #17

-fin-